Efficient Evaluation of Data-intensive Batch-queries in Open Simulation Laboratories

by

Kalin Nikolaev Kanov

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

June, 2015

Abstract

Better instruments, faster and bigger supercomputers and easier collaboration and sharing of data in the sciences have introduced the need to manage increasingly large datasets. Advances in high-performance computing (HPC) have empowered many science disciplines' computational branches. However, many scientists lack access to HPC facilities or the necessary sophistication to develop and run HPC codes. The benefits of testing new theories and experimenting with large numerical simulations have thus been restricted to a few top users. In this dissertation, I describe the "remote immersive analysis" approach to computational science and present new techniques and methods for the efficient evaluation of scientific analysis tasks in analysis cluster environments.

I will discuss several techniques developed for the efficient evaluation of data-intensive batch-queries in large numerical simulation databases. An I/O streaming method for the evaluation of decomposable kernel computations utilizes partial-sums to evaluate a batch-query by performing a single sequential pass over the data. Spatial filtering computations, which use a box filter, share not only data, but also compu-

ABSTRACT

tation and can be evaluated over an intermediate summed volumes dataset derived from the original data. This is more efficient for certain workloads even when the intermediate dataset is computed dynamically. Threshold queries have immense data requirements and potentially operate over entire time-steps of the simulation. An efficient and scalable data-parallel approach evaluates threshold queries of fields derived from the raw simulation data and stores their results in an application-aware semantic cache for fast subsequent retrieval. Finally, synchronization at a mediator, task-parallel and data-parallel approaches for the evaluation of particle tracking queries are compared and examined.

These techniques are developed, deployed and evaluated in the Johns Hopkins Turbulence Databases (JHTDB), an open simulation laboratory for turbulence research. The JHTDB stores the output of world-class numerical simulations of turbulence and provides public access to and means to explore their complete space-time history. The techniques discussed implement core scientific analysis routines and significantly increase the utility of the service. Additionally, they improve the performance of these routines by up-to an order of magnitude or more when compared with direct implementations or implementations adapted from the simulation code.

Primary Reader: Randal Burns

Secondary Readers: Alex Szalay & Charles Meneveau

Acknowledgments

Completing a Ph.D. dissertation takes more than the hard work and effort of a single individual. It takes nurturing a curious mind, inspiration to discover and explore the unexplored, support and encouragement to remain focused and think big, and instruction and guidance to learn new things and to keep improving. I am fortunate enough to have been surrounded by wonderful people that have been there throughout my life's journey and have provided all of these ingredients to make this dream of mine come true. For that I am sincerely grateful!

First and foremost, I would like to thank my advisor, Randal Burns for taking me under his wing, making me part of an exciting project, giving me the freedom to chart my own course, supporting me along the way, cultivating my analytical abilities and through instruction and mentorship turning me into a critical thinker. I am also fortunate to have worked with a great group of people, current and former members of the Turbulence Database Group and Johns Hopkins. I would like to thank Charles Meneveau, Alex Szalay, Gregory Eyink, Cristian Lalescu and Huidan Yu for their invaluable assistance, advice and counsel.

ACKNOWLEDGMENTS

The Computer Science Department at Johns Hopkins has been my home away from home for the past several years. Its exceptional faculty, staff and students have challenged me intellectually, supported me in many different ways and have shaped my life as a graduate student. I would like to thank Misha Kazhdan, Yanif Ahmad and Yair Amir for sharing their knowledge and their willingness to offer advice. My colleagues at the Hopkins Storage Systems Lab have engaged me in many interesting conversations and discussions over the years on topics ranging from research and computer science, to sports, music, food, travel and even airline frequent flyer programs. I would like to thank Eric Perlman, Osama Khan, Xiaodan Wang, Neal Walfeld, Priya Manavalan, Paul Stanton, Da Zheng, Kunal Lillaney, Disa Mhembere, Alex Baden and Stephen Hamilton for their collaboration, help with my research and for providing the needed occasional distractions. I would also like to acknowledge the support of the Computer Science Department staff: Debbie DeFord, Laura Graham, Javonnia Thomas, Cathy Thornton, Steve Rifkin and Steve DeBlasio.

I wouldn't be what I am today without the inspiration and influence of my close friends, who have often given me the extra push that I have needed and have always believed in me. Thank you to Hristo Ninov, Dimitar Dimitrov, Milen Bortchev, Ivailo Lozanov, Stanislav Harizanov, Raya Lakova and Pete Meliagros for being there when I have needed you.

I would also like to express my deepest gratitude to my family. Thank you for leaving footsteps for me to follow on the path of learning and exploration. Thank

you for sparking my interest in literature and science to make the journey stable and steady. Thank you for giving me the independence to make the hard decisions at the crossroads. And thank you for the unwavering support along the way. Thank you to my mother, Tsvetana Valkov, for the love, caring and the little bits of magic that you have always created in my life. You are my hero! Thank you to my father, Nikolay Valkov, for making me always strive to be as good as I can be. Thank you to my sister, Vera Kanova, for being my rock, my friend and my inspiration. Thank you to my grandparents, Kano and Vera Kosturkov, Peio and Stanka Tomov, for being my shining examples. Thank you to my uncle, Stanimir Staikov, for all the books and more that you have given me over the years.

Last but not least, I would like to thank the love of my life, my wife Teodora Mihaylova. Thank you for being my copilot, for being there through think and thin, for being my source of inspiration and motivation and for your patience and wisdom. It is your friendship that guides me as I learn and grow and it is your partnership that will lead me to the fulfillment of my dreams.

Dedication

To my family, my friends and my love ...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Most, if not all, science disciplines have developed computational branches. Examples include computational physics, computational biology, computational chemistry, computational fluid dynamics, and others. In some cases experimentation is simply not possible as is the case in cosmology. In others experimentation may be too expensive or too dangerous. In the case of fluid dynamics and turbulence research in particular direct numerical simulations (DNSs) are widely adopted tools for the development and refinement of turbulence models, which improve our understanding of how the underlying physical processes work. The Navier-Stokes equations, which govern turbulent fluid flow, are discretized and integrated forward in time, solving for physical field variables (e.g. velocity and pressure) as function of space and time in the domain of the simulation.

The memory and computational requirements of DNSs of turbulence scale as the

$\frac{9}{4}^{th}$ and third power of the non-dimensional Reynolds number, Re [1, 2]. Turbulent flows of practical interest usually have high Re, and must therefore be solved using supercomputers. Preparing and running such a simulation requires substantial expertise in parallel computing and turbulence research, but moreover it requires access to a supercomputing facility. Traditionally, such undertakings have been team efforts that are planned and prepared for ahead of time, the particular science questions to be answered have been thought of in advance and most of the analysis is done during the simulation. Only representative snapshots are stored for subsequent analysis and even if more data are available, accessing them is often not easy and usually entails the downloading of large amounts of data to a local machine. As a result, the same simulations must be repeated after new questions arise that were not initially obvious; most breakthrough concepts cannot be anticipated in advance. In order to provide access to high-quality world-class turbulence DNS data to researchers that may not have access to supercomputing facilities and the public as a whole, our group at Johns Hopkins has developed an open turbulence laboratory, the Johns Hopkins Turbulence Databases (JHTDB) [3].

1.1 The JHTDB, an Open Simulation Laboratory for Turbulence Research

In this dissertation, I describe our experience building and operating the JHTDB and our efforts to provide easy access to terabyte simulation datasets to anyone in the world with an Internet connection. An open simulation laboratory complements the DNS approach to the study of turbulence. The entire space-time histories of landmark simulations are stored persistently in a database cluster. Armed with the JHTDB's built-in analysis functionality, researchers with modest computational and storage capabilities can perform sophisticated analysis using high-resolution datasets, traverse the flow forward and backward in time, repeat and refine experiments, tune the parameters of their analysis routines, test new hypotheses, and use the data in ways that weren't even envisioned when the simulations were first run. Additionally, an open simulation laboratory can be used for teaching and training the next generation of researchers.

1.1.1 Exploration of Turbulence Simulation Data

The archival approach to numerical simulation data has several advantages when compared to hypothesis-driven simulation, in which a code is run every time a new question emerges. One, it reduces the time from hypothesis to drawing conclusions. Analysis queries are executed against stored data, and, in many cases, have much

smaller data and computation requirements when compared to an entire simulation. Two, it enables data exploration. The same points in space and time can be explored multiple times as intuition about the data develops. Three, it provides for a seamless verification of experimental results. Analysis queries used to draw conclusions can be easily rerun and are guaranteed to produce the same results. There are almost no drawbacks when an experiment fails and needs to be repeated with a different set of assumptions, initial conditions or parameters. Finally, it does not assume that an entire dataset or a significant fraction of it is downloaded, forcing the user to provide the computational resources for analysis of multi-terabyte/petabyte datasets.

The deployment of an open simulation laboratory for turbulence research has already transformed our understanding of turbulence and has allowed for new types of analysis to be performed. Users can track particles immersed in the simulated flow both forward and backward in time, evaluate ensembles of particles together, and repeat experiments to achieve confidence in the results. Researchers at Johns Hopkins performed this type of analysis for magnetohydrodynamics (MHD), tracing stochastic trajectories arriving at a point $(\mathbf{x}, t)$ backward in time to the initial time, t_0 and then transporting the magnetic field forward along the trajectories to the point $(\mathbf{x}, t)$. This numerical experiment has revealed why magnetic field-lines in solar flares break and reconnect in as little as 15 minutes as opposed to the millions of years predicted by classical theory [4]. The original numerical simulation was performed years ago to explore an unrelated hypothesis!

```
% Requesting vorticity threshold.
threshold_array =  getThreshold (authkey, 'isotropic1024', 'vorticity', ...
    time, 65.0, FD4NoInt, X, Y, Z, Xwidth, Ywidth, Zwidth);
% Determine maximum value of the vorticity.
[M,I] = max(threshold_array(4,:));
% Set domain size and position for JHTDB request.
nx = 128; ny = nx;
xoff = (threshold_array(1,I) - nx/2)*spacing;
yoff = (threshold_array(2,I) - ny/2)*spacing;
zoff = threshold_array(3,I);
npoints = nx*ny;
% Create surface.
x = linspace(0, (nx-1)*spacing, nx) + xoff;
y = linspace(0, (ny-1)*spacing, ny) + yoff;
[X Y] = meshgrid(x, y);
points(1:2,:) = [X(:)'; Y(:)'];
points(3,:) = zoff*spacing;
% Get the velocity gradient at each point.
gradient = getVelocityGradient (authkey, 'isotropic1024', time,
    FD4Lag4, NoTInt, npoints, points);
% Calculate vorticity magnitude.
vort = [gradient(8,:) - gradient(6,:); gradient(3,:) - ...
    gradient(7,:); gradient(4,:) - gradient(2,:);];
vort_mag = sqrt(vort(1,:).^2 + vort(2,:).^2 + vort(3,:).^2);
VortMag = transpose(reshape(vort_mag, nx, ny));

% Plot vorticity magnitude contours.
contourf(X, Y, VortMag, 30, 'LineStyle', 'none');
set(gca, 'FontSize', 11)
title('Vorticity Magnitude', 'FontSize', 13, 'FontWeight', 'bold');
```

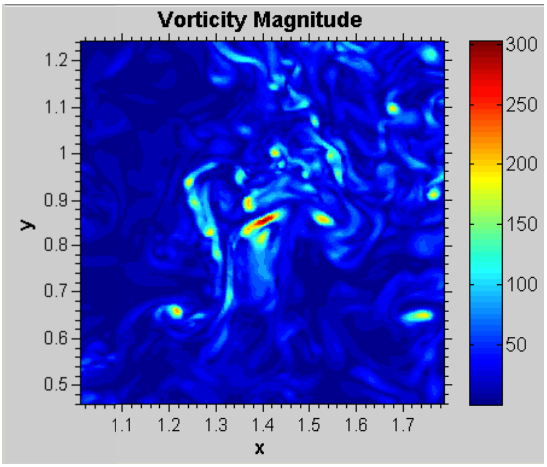


Figure 1.1: Matlab script to examine the vorticity magnitude around a point with large vorticity in the isotropic turbulence dataset stored in the JHTDB.

1.1.2 Immersive Turbulence

Using the JHTDB and accompanying access libraries and tools, scientists can immerse virtual sensors in the simulation, visualize turbulent phenomena, perform feature extraction and data mining. The data and the associated functionality are available on-demand and over the Internet. This allows scientists to interact with the entire simulation data. Stationary sensors immersed in the flow report the simulation parameters or values of derived fields at particular locations, while particles can be allowed to flow both forward and backward in time along pathlines in the entire temporal domain or along streamlines within an individual time-step. We term this approach to computational science "remote immersive analysis".

Figure 1.1 shows an example of the remote immersive analysis approach to computational turbulence. The user extracts all locations within the region of interest where the magnitude of the vorticity is above a prescribed threshold (65.0 in the example) from the isotropic turbulence dataset (making a call to the getThreshold Web-service method). He or she then generates a 2D planar patch with the location of maximum vorticity in the center and places "virtual sensors" at each location using the dataset's grid spacing. He or she then requests the velocity gradient (making a call to the getVelocityGradient Web-service method) at each of these locations and computes and plots the vorticity magnitude to yield the visualization shown. The results of the getThreshold and getVelocityGradient methods can be further analyzed depending on the particular needs of the user.

Many numerical simulations are canonical and have useful lifetimes of years or decades. Turbulence researchers have come to rely on the JHTDB to provide such canonical datasets and new ideas are tested in this laboratory routinely. The scientific results that have emerged span the full breadth of turbulence research, from physical theory (e.g. turbulent dynamics of velocity gradients [5]), to engineering modeling (e.g. large-eddy simulation of wall-bounded flows [6]), to development of experimental techniques (e.g. particle-based measurements of very fine-scale flow structure [7]). These are just a handful of the many examples showing the utility of the JHTDB and how it has enabled data-driven discovery.

1.1.3 Challenges

The great advances in computing capabilities of the recent past have led to scientific datasets of unprecedented size. The standard DNSs of turbulent flows produce datasets that are 10s to 100s of terabytes, while leading-edge simulations have already crossed the petabyte threshold [2, 8]. The rate at which new datasets are generated is also accelerating. This highlights the enormity of the task at hand; to persistently store large datasets, provide public access to them and ensure that they can be analyzed in an efficient manner.

1.2 Summary of Contributions

In order to support the exploration of large numerical simulation datasets and make scientific analysis of the data possible over the Internet, my collaborators and I have developed efficient methods for the evaluation of batch-queries of several different types. In some cases the methods that we have developed make the execution of certain scientific analysis tasks practical where it was not before. In other cases they improve the performance of these tasks by up to an order of magnitude or more. In this dissertation, I describe these system techniques and present a comprehensive evaluation of the different methods in a live production environment. I present and discuss the following contributions:

- **I/O streaming evaluation of batch-queries performing decomposable kernel computations.** We have developed a method, which evaluates batch-queries performing kernel computations at a large set of target locations by means of a single sequential pass over the data. The method improves the performance of queries performing interpolation, differentiation, filtering and other kernel computations by over an order of magnitude when compared with direct evaluation of these computations at each target location or execution strategies adapted from the simulation code.

- **Data-driven evaluation of decomposable kernel computations by means of partial-sums.** The I/O streaming method that we have developed relies on the fact that a linear-sum computation over a kernel of data points can be broken down into partial-sums. The data requirements of individual requests are identified and merged in a preprocessing step. As data chunks (also known as *atoms*) are retrieved from the database a partial-sum computation is executed over the data points in the overlapping region between a request's kernel of computation and the retrieved data region. Partial results are maintained and updated as more of the data become available. This allows us to retrieve the data in any order and in parts.

- **Distributed evaluation of batch-queries.** Evaluating by partial-sums provides support for distributed evaluation of batch-queries. Each database process

performs a partial evaluation over the data that it has available locally and partial results are merged at the mediator.

- **Spatial filtering by means of summed-volumes.** We have developed a complementary method for the evaluation of the largest of spatial filtering queries, those that perform filtering with large filter widths over locations densely clustered in space. The method dynamically computes an intermediate summed-volumes dataset over the bounding data region of the set of input points and evaluates each spatial filter by looking up only 8 data points in this summed-volumes dataset.

- **Query-processing framework for spatial-filtering queries.** The query-processing framework that we have built dynamically decides between I/O streaming and summed-volumes evaluation of spatial-filtering queries based on the workload's characteristics, such as density of target locations and expected number of I/Os.

- **Computing derived fields of large simulation data on-demand and evaluating threshold queries on them at extreme scale.** Evaluating threshold queries of derived fields provides large data analytics capabilities that examine the entire stored data volume of simulation time-steps. This is achieved through the combination of existing data management techniques such as data parallelism and semantic caching as well as taking advantage of heterogeneous

scientific cluster architectures (sharded relational DBMS with several SSDs per node).

- **Study of particle tracking techniques in an open simulation laboratory.** We examine task-parallel, data-parallel and mediator synchronization techniques for the advection of particles in the simulated flow with a focus on the I/O demands of the process.

- **Task-parallel method for the advection of particles.** We have developed a task-parallel method for particle integration in a scientific database cluster environment. The method improves performance by up to a factor of 3 and achieves scalability by performing the entire processing on the database nodes of the cluster.

- **Comparative study of the trade-offs between batch execution and asynchronous processing.** We show that the reduced I/O time from the retrieval of data for an entire batch of particles outweighs the advantages of asynchronous processing.

- **Experimental evaluation on data-intensive workloads in a live production environment.** We evaluate the different methods on workloads extracted from the usage logs of the JHTDB and on micro-benchmarks defined based on common query patterns and designed to examine the scalability properties of each method. The evaluation is done in the live production environment of the

JHTDB and we show scalability results on datasets hundreds of terabytes in size.

1.3 Dissertation Outline

The remainder of the dissertation is organized as follows. In chapter 2, I provide an overview of the JHTDB, its capabilities and the functionality that it provides. Chapter 3 describes the I/O streaming method for the evaluation of batch-queries that perform kernel computations. Chapter 4 describes the summed-volumes method for the evaluation of spatial filtering queries and the query execution framework, which dynamically decides between I/O streaming and summed-volumes based on the workload characteristics. Chapter 5 describes the framework for evaluation of threshold queries of derived fields and the application-aware semantic cache for the results of these queries. Chapter 6 discusses particle tracking in the JHTDB and evaluates several approaches for particle advection. In chapter 7, I present a summary of the impact of the JHTDB on the study of fluid dynamics and turbulence research. Finally, Chapter 8 concludes this dissertation with an overview of the contributions and a discussion of future work directions.

Chapter 2

Background

The JHTDB was designed to provide efficient yet flexible access to world-class high-resolution turbulence DNS data [9, 10]. Over the past several years since its inception in 2007 some aspects of the architecture were modified, new analysis functionality was added and new ways of accessing the data were provided. I provide an overview of the architecture and summarize some of these developments here.

I describe two versions of the JHTDB that define the *immersive turbulence* approach (Section 1.1.2), in which we store the output of world-class, high-resolution simulations in a cluster of relational databases and allow scientists to explore, analyze, and visualize the data through Web-services. The first version of the database hosts 27 TB of publicly-available data from a direct-numerical simulation (DNS) of forced isotropic turbulence over 1024 time-steps on a 1024^3 grid [9]. My collaborators and I have explored some of the design flaws and limitations of the initial design, which

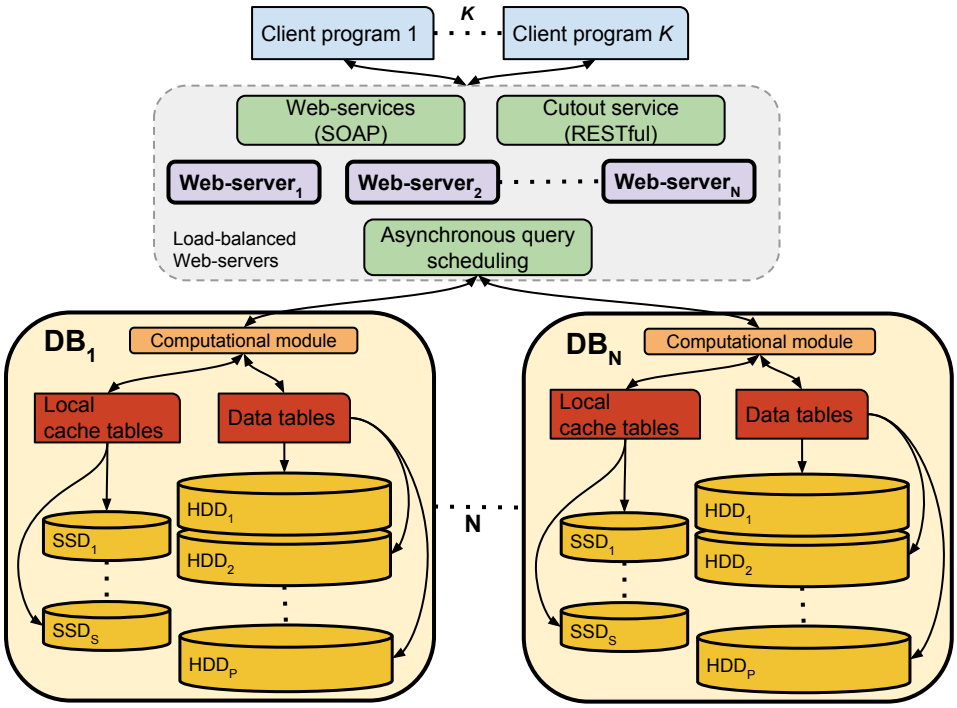


Figure 2.1: Architecture of the JHTDB.

have motivated the current line of research. I also discuss the second version of the JHTDB and the evolution of the service, which now stores three additional datasets, provides additional functionalities and has increased capabilities.

2.1 Architecture of the JHTDB

In both versions of the JHTDB, data are partitioned spatially and temporally across the database cluster and are accessed through a Web server. The Web server acts as a mediator that divides user requests according to the spatial partitioning of the data and submits them for execution to the appropriate database nodes (Figure 2.1). Li et al. [9] and Perlman et al. [10] describe the architecture in detail.

To amortize I/O, improve the specificity of reads, and reduce the overhead of data indexing, we partition the data into data cubes, or "atoms," which are represented as binary-large objects (or BLOBs) in the database tables. Each atom stores 8^3 data points and stores the data for one simulation field (e.g. velocity or pressure). Most of the computations require data from the target location's immediate neighborhood as a kernel (such as a cube centered on the target location). Each kernel's size depends on the request's parameters, but it is usually in the 4^3 to 8^3 data point range. Given a size of 8^3 data points for a data atom, each record fits within a single database page, which helps us obtain each computation's data with a small number of I/Os. Additionally, indexing each individual point would nearly double the storage requirements.

The Morton z-order space-filling curve governs the spatial partitioning and organization of the data. The curve provides the mapping of the three-dimensional data to the linear ordering in which the atoms are laid out on disk. The Morton z-order curve was chosen because it is admissible, it provides a stable ordering of space, the indexes are easy to compute and it preserves spatial locality well, which is important because data points in close proximity in physical space are usually accessed at the same time in the database's typical usage patterns. The spatial access method used to store and retrieve the data atoms to and from disk is a standard B+ tree clustered index, which is keyed with a combination of the time-step and the Morton z-curve index.

Option	Description
NoSInt	No Space interpolation (value at the datapoint closest to each coordinate value)
Lag4	4th-order Lagrange Polynomial interpolation along each spatial direction
Lag6	6th-order Lagrange Polynomial interpolation along each spatial direction
Lag8	8th-order Lagrange Polynomial interpolation along each spatial direction
M1Q4	Splines with smoothness 1 (3rd order) over 4 data points.
M2Q8	Splines with smoothness 2 (5th order) over 8 data points.
M2Q14	Splines with smoothness 2 (5th order) over 14 data points.
FD4	4th-order centered finite differencing (can be spatially interpolated)
FD6	6th-order centered finite differencing (without spatial interpolation)
FD8	8th-order centered finite differencing (without spatial interpolation)

Table 2.1: Spatial interpolation and differentiation options in the JHTDB.

2.2 Spatial and Temporal Interpolation

Accurate spatial and temporal interpolations are important to obtain simulation parameters off grid nodes. The JHTDB implements spatial interpolation using Lagrange and spline polynomials with various optional orders of accuracy as well as temporal interpolation using Picewise Cubic Hermite Interpolation Polynomials (PCHIP) [11]. Table 2.1 shows a list of the available interpolation and differentiation options. I briefly describe the Lagrange polynomial and PCHIP interpolation techniques and their data requirements here. More details are provided in Li et al. [9] and on the project's Website [12].

The interpolated value of a variable at a target location that does not coincide with a grid node can be computed using Lagrange polynomial interpolation. Lagrange polynomial interpolation operates over a cubic region surrounding the target location. This region is the kernel of computation and the data values at all grid

locations within that region are needed for the computation of the interpolated value. The interpolated value of any point in space $\mathbf{x}'$ computed using N^{th} order Lagrange polynomial interpolation is given by:

$$f(\mathbf{x}') = \sum_{k=1}^{N}\sum_{j=1}^{N}\sum_{i=1}^{N} l_z^{q-\frac{N}{2}+k}(z') l_y^{p-\frac{N}{2}+j}(y') l_x^{n-\frac{N}{2}+i}(x') \cdot f(x_{n-\frac{N}{2}+i}, y_{p-\frac{N}{2}+j}, z_{q-\frac{N}{2}+k}) \,, \tag{2.1}$$

in which $\mathbf{x}' = (x', y', z')$ is the position of the target point in 3-dimensional space and $f(x_i, y_j, z_k)$ represents the data stored at the node on the grid at location (x_i, y_j, z_k). The computation kernel is an $N \times N \times N$ cube around the target point. The location $(x_n,\, y_p,\, z_q)$ on the grid is computed as $n = \left\lfloor \frac{x'}{\Delta x} + \frac{1}{2} \right\rfloor$, $p = \left\lfloor \frac{y'}{\Delta y} + \frac{1}{2} \right\rfloor$, $q = \left\lfloor \frac{z'}{\Delta z} + \frac{1}{2} \right\rfloor$, where $\Delta x,\, \Delta y,\, \Delta z$ are the widths of the grid in the $x,\, y$ and z dimensions. The Lagrange coefficients l are given by:

$$l_\theta^i(\theta') = \frac{\prod_{j=\alpha-\frac{N}{2}+1, j\neq i}^{\alpha+\frac{N}{2}} (\theta' - \theta_j)}{\prod_{j=\alpha-\frac{N}{2}+1, j\neq i}^{\alpha+\frac{N}{2}} (\theta_i - \theta_j)} \,, \tag{2.2}$$

in which θ can be $x,\, y$ or z and α can be $n,\, p$ or q, respectively.

Similarly to spatial interpolation, temporal interpolation can be used to obtain the value of a variable in-between stored time-steps with higher accuracy. Temporal interpolation can be performed using PCHIP. In the JHTDB, the value from the two

nearest time-steps is interpolated at time t' with centered finite difference evaluation of the end-point time derivatives. A total of four temporal points are used:

$$f(t') = a + b(t' - t_n) + c(t' - t_n)^2 + d(t' - t_n)^2(t' - t_{n+1}) \ , \tag{2.3}$$

in which $f(t_n)$ denotes the data stored at time t_n, $n = \left\lfloor \frac{t'}{\Delta t} + \frac{1}{2} \right\rfloor$ with Δt being the time increment between consecutive times stored in the database and a, b, c and d are given by:

$$\begin{aligned} a &= f(t_n) \\ b &= \frac{f(t_{n+1}) - f(t_{n-1})}{2\Delta t} \\ c &= \frac{f(t_{n+1}) - 2f(t_n) + f(t_{n-1})}{2\Delta t^2} \\ d &= \frac{-f(t_{n-1}) + 3f(t_n) - 3f(t_{n+1}) + f(t_{n+2})}{2\Delta t^3} \ . \end{aligned}$$

2.3 Limitations and Lessons Learned

Our experience deploying the first version of the database revealed several design errors. We stored all attributes in a single table, which reduces I/O performance. For example, scientists would be interested in just one of the fields (velocity or pressure). However, since velocity and pressure were stored together in the same atom, data for both would be retrieved when each particular query was evaluated. We also chose

to replicate data on the edges of partitions in order to localize the computation of kernels to single database nodes. Since the highest order interpolation supported was 8^{th} order Lagrange interpolation, which has a kernel of size 8^3, the required replication was 4 data points in each dimension (a kernel half-width). This introduced $\sim$42% storage overhead.

2.4 Second Version of the JHTDB

The second version of the database amends these decisions in part due to the I/O streaming techniques we develop (Section 3). We employ vertical partitioning in order to improve the specificity of reads. The MHD data set consists of 3 vector fields, velocity, magnetic field, and magnetic vector potential, and the scalar pressure field. These data are partitioned into 4 tables respectively, and when a request for a particular field is made we only have to read the specific data instead of all of them. This vertical partitioning is reminiscent of column-store databases [13, 14, 15]. In order to reduce wasted data transfer and to improve memory performance, we reduce the size of a data atom to 8^3 (6144B of storage for vector fields, e.g. storing V_x, V_y and V_z together). Small data atoms do not pollute the cache and have small transfer times from disk that outweigh the potentially higher costs of more disk seeks. Evaluating by partial-sums allows us to distribute the computation for kernels that cross

node boundaries and work with smaller atoms eliminating the need for replication as described in Section 3.2.3.

2.4.1 Datasets

The JHTDB currently stores four datasets – the DNS output of four types of turbulent flows:

- a forced isotropic turbulence dataset on 1024^4 space-time data points at a Taylor-scale Reynolds number of $Re_\lambda = 433$;
- a steady-state incompressible magnetohydrodynamic (MHD) turbulence dataset on 1024^4 space-time data points, forced with a Taylor-Green flow, with $Re_{\lambda,u} = 186$ and $Re_{\lambda,b} = 144$ for the velocity and magnetic field, respectively;
- a channel flow dataset at $Re_\tau = 1000$ on $2048 \times 1536 \times 512$ spatial data points and 4000 time-steps;
- a variable-density mixing flow dataset on 1024^3 spatial data points and 1015 time-steps.

The total amount of space occupied by these datasets is over 200TB. The data are partitioned spatially and temporally across a cluster of relational databases. The database nodes are part of the 1.1 PB GrayWulf cluster [16] and the 11 PB DataScope cluster [17] at JHU. Each node runs Microsoft SQL Server and provides built-in analysis functionality, implemented as user-defined functions or stored procedures in the

Common Language Runtime (CLR). The data and analysis functionality are accessible through a Web-server front end (http://turbulence.pha.jhu.edu), which provides Web-services implemented using the SOAP protocol (Simple Object Access Protocol), a cutout service producing HDF5 files [18] of the raw simulation data and a Web-query interface. Users can access the Web-services directly or through several client libraries, which we have provided, including Matlab, C, Fortran and Python wrappers (see Figure 2.1).

2.4.2 Analysis Functionality

The analysis functions used in turbulence research are data-intensive; they operate over collections of points and to compute the result at each target point they access a large number of neighboring data points. Some functions are also data reducing; they evaluate or compute over a large number of data points, but produce smaller result sets. These types of computations are much more efficiently executed on the servers near the data. Large amounts of data do not have to be moved across the network and the computations can be evaluated in a distributed manner across the nodes of the cluster. However, the functionality provided to users should also be general enough to allow for different types of analysis and customization of experimental parameters. With these considerations in mind, we have developed Web-services methods that evaluate simulation parameters on a set of target locations. Users of the JHTDB can request the following:

- values of the simulation fields on and off the grid (locations off the grid are interpolated using Lagrange and spline polynomials of varying orders);
- first and second order derivatives of the simulation fields evaluated using finite-differencing schemes of varying orders;
- filtered quantities, such as the filtered velocity, filtered velocity gradient, sub-grid stress tensor, etc. (filtering is performed using a box filter);
- the positions of particles tracking the simulated flow (particle tracking is performed using a second order accurate Runge-Kutta integration scheme);
- all values of the simulation fields and derived quantities above a prescribed threshold (derived quantities currently include the vorticity and Q-criterion).

Table 2.2 lists the currently available functions in the JHTDB and the datasets to which they apply.

Function	Isotrpic	MHD	Channel	Mixing
GetVelocity	✓	✓	✓	✓
GetMagneticField		✓		
GetVectorPotential		✓		
GetPressure	✓	✓	✓	✓
GetVelocityAndPressure	✓	✓	✓	✓
GetForce	✓	✓		
GetVelocityGradient	✓	✓	✓	✓
GetMagneticFieldGradient		✓		
GetVectorPotentialGradient		✓		
GetPressureGradient	✓	✓	✓	✓
GetPressureHessian	✓	✓	✓	✓
GetVelocityLaplacian	✓	✓	✓	✓
GetMagneticFieldLaplacian		✓		
GetVectorPotentialLaplacian		✓		
GetVelocityHessian	✓	✓	✓	✓
GetMagneticFieldHessian		✓		
GetVectorPotentialHessian		✓		
GetPosition	✓	✓		✓
GetBoxFilter	✓	✓		✓
GetBoxFilterSGSscalar	✓	✓		✓
GetBoxFilterSGSvector	✓	✓		✓
GetBoxFilterSGSsymtensor	✓	✓		✓
GetBoxFilterSGStensor	✓	✓		✓
GetBoxFilterGradient	✓	✓		✓
GetDensity				✓
GetDensityGradient				✓
GetDensityHessian				✓
GetThreshold	✓	✓	✓	✓

Table 2.2: Functions supported in the JHTDB and the dataset to which they apply.

2.4.3 Cutout Service

The JHTDB also provides a cutout service, which allows users to obtain the raw simulation data in familiar HDF5 format [18]. Users can optionally specify a stride or a step, and a filter width. If these options are specified, the cutout service produces a coarser version of the data, and if the filter width is larger than a single data point the data are filtered using a box filter. For reasonable values of the stride parameter producing a filtered version of the data is only slightly slower than obtaining the raw data through strided access: I/O limits the data access and network speed limits the transfer of results to the user. The filtering computation introduces a small overhead mainly due to the fact that the data have to be converted from binary to single precision floating point format to perform the filtering computations.

Chapter 3

I/O Streaming Evaluation of Batch Queries

Many scientific analysis queries perform linear sum computations or kernel computations, which operate on a neighborhood of data points. Examples include interpolation, differentiation and filtering. These routines are fundamental to computational turbulence and over 95% of the analysis queries submitted to the JHTDB make use of such routines. In this chapter, I describe a method for evaluating computational turbulence queries, including Lagrange Polynomial interpolation, based on partial-sums that allows the underlying data to be accessed in any order and in parts [19]. The method exploits these properties to stream data from disk in a single pass and concurrently evaluates batch queries. The combination of sequential I/O and data sharing improves performance by an order of magnitude when compared with di-

rect evaluation of each query. The technique also supports distributed evaluation of queries in a database cluster, assembling the partial-sums from each node at the query mediator. The I/O streaming partial-sums method allows the JHTDB to realize scale and throughput for our scientists' data-intensive workloads.

3.1 Motivation

Data-intensive computing has revolutionized access to the computational simulation of turbulence. As described in Section 2 we have built the JHTDB in order to provide public access to world-class numerical simulations of turbulence. However, a clustered database approach to computational turbulence has revealed performance issues for data-intensive workloads. Query throughput and system scalability limit the utility of the service, restricting the number of concurrent queries and increasing the time needed to complete experiments. Heavy usage can slow down the service by a factor of 10 to 20. The first iteration of query evaluation techniques [9] were adapted from the simulation code and exhibit poor access locality at all levels of the memory hierarchy and incur substantial storage overhead associated with the replication of data across nodes (see Section 2.3).

The predominant turbulence analysis query performs a kernel computation, such as spatial interpolation of a vector field at a specific point based on high-order Lagrange Polynomials. To evaluate each point a kernel computation (Figure 3.1) uses

data points from the simulation, e.g. 8^3 data points for 8^{th} order Lagrange Polynomial interpolation. Typically, scientists request batches of point queries with varying sizes (thousands to hundreds of thousands or even millions of point queries) from the same simulation time-step. Evaluation of these queries consists of embedded loops that iterate over the kernel of computation. This does not access data coherently; it retrieves data from the same cache line or page in multiple loop iterations. It also accesses the same data values multiple times when the kernels of multiple queries overlap.

To improve memory and I/O performance, we define a data-driven partial-sums method for the concurrent evaluation of multiple kernel computation queries in a single, streaming pass over the data. The queries can be evaluated incrementally and in parts as kernel computations operate over a linear combination of data values and coefficients. As an example, Lagrange interpolation is a linear combination of the data values and the Lagrange polynomials, which only depend on the target position and the resolution of the grid. Therefore, we perform the computation for all queries at the same time by maintaining a partial-sum for each target point and access the data sequentially. This makes memory accesses coherent in all levels of the cache hierarchy, regardless of the cache line size. Sequential access patterns use the full parallelism of the memory hardware, avoid associativity or bank conflicts, and make processor, I/O, and database prefetching effective. Most importantly, I/O streaming reduces the overall I/O requirements. Batches of concurrent queries share I/O; a single value from

the database contributes to all queries for which the kernel of computation contains the point.

Evaluating by partial-sums supports distributed evaluation and eliminates the need for data replication among database nodes. Previous approaches have replicated a kernel half-width at the boundaries of a partition in order to avoid transferring data among nodes (Section 2.3). When a computation kernel spans data partitions, the JHTDB mediator splits a single query into two or more partial-sum queries to the multiple database nodes and combines the results. Distributing queries has minimal performance overhead and reclaims the 42% space used for replicated data that localized queries to single nodes [10]. It also supports arbitrarily large kernels; replication restricted us to 8^{th} order Lagrange interpolation previously.

I/O streaming and evaluating by partial-sums applies to a broad class of decomposable functions in the realm of data-intensive science. This includes all computations that are expressed as a linear combination of the data samples, such as differentiation, integration, and filtering. We have implemented I/O streaming and distributed evaluation in the second generation Turbulence Database Cluster (Section 2.4). We have evaluated our technique on user workload traces from the JHTDB. We have realized a speedup of at least 6 times and up to 50 times in the overall performance of queries that compute Lagrange interpolation when compared with a direct method of evaluation. Since Lagrange interpolation is the most commonly used function this translates into improved performance for scientific experiments. Furthermore, I/O

streaming and evaluating by partial-sums is used in the JHTDB for the evaluation of spline polynomial interpolation, first and second order differentiation, filtering and particle integration.

3.2 Data-driven Query Execution

My collaborators and I have implemented a data-driven partial-sums method for the computation of decomposable kernel computations, including high-order Lagrange interpolations in the JHTDB. We perform the computations by means of an I/O stream that takes a single pass over the data. For each target point, we maintain a partial-sum of the results that we update incrementally as we access relevant data points. We also use partial-sums to implement distributed evaluation of interpolation kernels across multiple database nodes. Each database node executes its part of the computation and maintains a partial-sum. The mediator combines the partial-sums and returns the final result to the user.

3.2.1 Partial-Sums

I described the evaluation of a point query by means of partial-sums in the context of Lagrange polynomial interpolations. However, this technique applies to any decomposable kernel computation that uses a linear sum combination of data and co-

efficients, including nested computations such as interpolating the value of a derivative obtained using a finite differencing scheme.

A decomposable kernel computation over grid data is one that consists of linear combinations of the values at grid nodes. In one dimension it is given by:

$$g(x') = \sum_{i=1}^{N} l_i(x') \cdot f(x_{n-\frac{N}{2}+i}) \tag{3.1}$$

in which the kernel of computation is of size N, the data are stored at discrete locations x_i on the grid (integer i), the location x_n is the one closest to x' and l_i are coefficients that do not depend on the data, but can depend on the target location x' and the grid resolution.

Given P, a permutation of the set $(1, ..., N)$, and a set $(S_1, ..., S_m)$ of contiguous non overlapping subsets of P, a decomposable kernel computation can be broken down into partial-sums as follows:

$$g(x') = \sum_{j=1}^{m} g_j(x') \tag{3.2}$$

in which g_j are the partial-sums over the sets S_j:

$$g_j(x') = \sum_{i \in S_j} l_i(x') \cdot f(x_{n-\frac{N}{2}+i}). \tag{3.3}$$

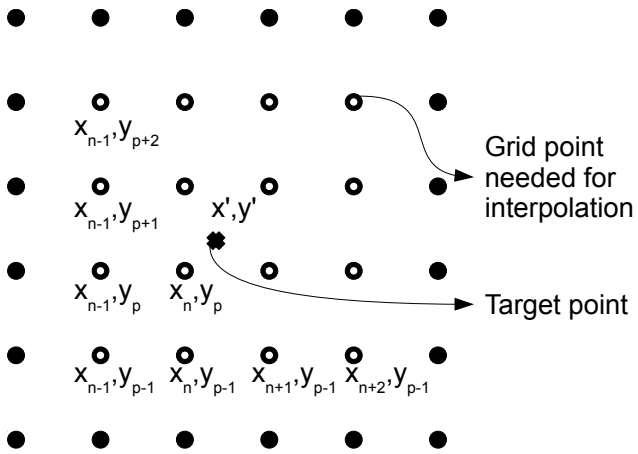


Figure 3.1: Data requirements for Lagrange Polynomial interpolation at a target point (x', y'). A 4^{th} order interpolation in two dimensions (shown) accesses a square of size 4 around the target point.

This allows us to perform incremental evaluation if only a portion of the data are available.

As described in Section 2.2, Lagrange polynomial interpolation uses a cubic grid of data surrounding the target point as the kernel of computation. The computation requires the data from an $N \times N \times N$ cube around the target point (Figure 3.1). We observe that Equation 2.1 can be computed incrementally and in parts. It is a linear combination of the data values and the coefficients $l^i_\theta(\theta')$. The coefficients are independent of the data values at grid nodes.

We leverage partial-sums to evaluate multiple point queries concurrently, using a single, sequential pass over the data. For each data atom, we update all of the interpolation kernels that include data from it, evaluating the portion of the computation contributed by the data points $f(x_{n-\frac{N}{2}+i}, y_{p-\frac{N}{2}+j}, z_{q-\frac{N}{2}+k})$ (Equation 2.1) that are part of this atom. We allocate space for the partial-sum and compute the Lagrange

coefficients on the first data point in the kernel. Interpolation queries remain active until the last data access.

We compute the Lagrange polynomial coefficients efficiently based on Purser and Leslie's procedures [20]. Purser and Leslie observed that the values in the denominator of Equation 2.2 are constant and do not depend on the coordinates of the target point. Thus, the values in the denominator are pre-computed and reused for the entire batch. Also, careful coding of the formulas eliminates redundant multiplications. These two optimizations reduce the time complexity of the computation of the coefficients to $O(N)$ from $O(N^3)$. In our system, these techniques result in small performance gains, because I/O, not computation, bounds data-intensive workloads.

3.2.2 I/O Streaming

Direct approaches to the evaluation of turbulence queries produce cache faults and perform redundant I/O, accessing the same data multiple times. These approaches gather the data points in the kernel of a given query in their entirety before evaluating the interpolation function. Kernels that span data atoms (partitions) perform multiple I/Os to collect the kernel data. As many as 27 I/Os are needed for points that span three atoms in all three dimensions given atom width of 4 data points in each dimension. These I/Os produce seeks in the disk system and perform incoherent accesses in cache memories, making unaligned requests for small amounts of data in each cache line. Atoms that cover multiple kernels are read multiple times by different

queries. This results in cache misses at all levels in the hierarchy, including to disk. The Morton z-order (and other space-filling curves) cluster data well. However, no linear ordering of data localizes all computations, because the data are 3-dimensional and partitioned.

Evaluating by partial-sums allows us to stream I/O, performing a sequential scan of the data that reads each data atom only once. We execute a multi-point query (or a batch query) as follows. For each query, we create hash table entries for each of the data atoms needed by the query's interpolation kernel. The hash table is keyed by the Morton z-curve index of the data atoms and looks up the partial-sum of the query. Thus, each hash table entry contains the list of point queries that need to be updated when reading the corresponding data atom. We maintain a small amount of metadata for each query in addition to the partial-sum, including the number of data atoms that have been read, the total number of atoms required, and the Lagrange coefficients needed for interpolation. To retrieve data, we build a temporary table that stores the z-indexes of the needed data atoms and perform a join between this temporary table and the data table. The database chooses its preferred I/O plan for this join, which always accesses the underlying data table sequentially.

The results of this join form an I/O stream on which we evaluate partial-sums. As a data atom arrives, we perform a hash table lookup to determine all queries that require points within the atom and update their partial-sums (Figure 3.2). The first time we update a partial-sum for a query, we compute the Lagrange coefficients for

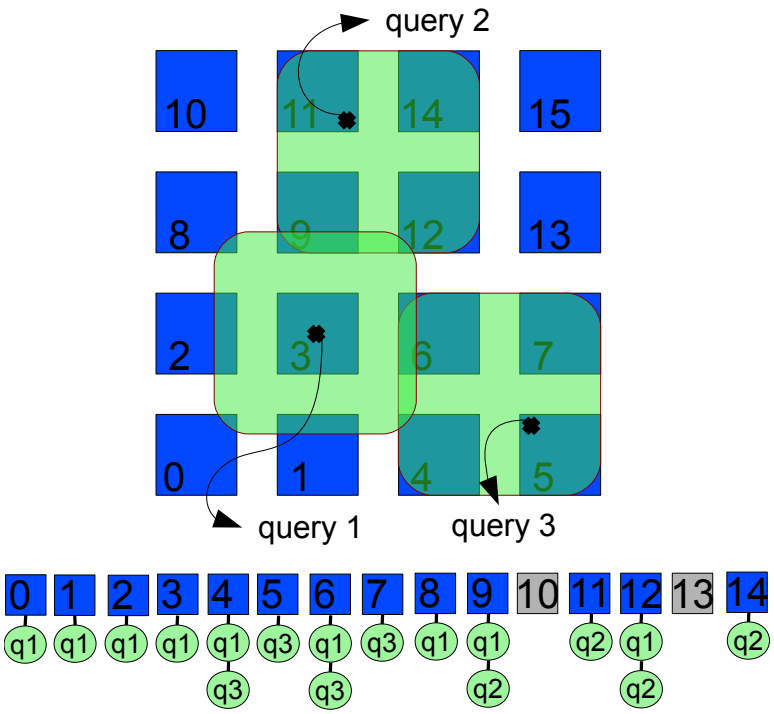


Figure 3.2: Processing a batch query. The kernel computation of each query is shown in light green. The data atoms (blue) that are intersected by the kernel of each query need to be accessed. The arrival order of each atom and the query that needs data from it is shown on the bottom, atoms in gray (10 and 13) are not needed and are not read.

that target point. We cache these coefficients for reuse on subsequent partial-sums. This is a time/space tradeoff; the coefficients could be recomputed for every data atom were the system memory constrained. When all of the needed data atoms have been processed, we return the result to the mediator.

The benefits of I/O streaming come at the expense of modest memory consumption. For each query, we create a hash table entry for every data atom it accesses, typically around 20 per query. Other metadata for the queries, including the partial-sum and the Lagrange coefficients, are allocated dynamically and only retained between

the first and last update of its partial-sum. Our evaluation shows that I/O streaming requires only tens to hundreds of megabytes of memory.

We sort the temporary table for batches smaller than 100,000 queries and when more than 1.1 queries access each data atom on average. In these cases, the database chooses a nested loops join and performance improves when both relations are sorted on the join key. Sorting larger batches has a negative impact on query performance, because the database chooses a sort merge join, which sorts the temporary table again.

Our final optimization performs loop unrolling to ensure that data are accessed sequentially within each cache line. The original direct computation of Lagrange interpolation loops over the x then y then z dimensions for the entire kernel. Because vector components are stored as tuples in row major order, e.g. velocity is $\langle V_x, V_y, V_z \rangle$, these loops perform strided access to individual velocity components, which reduces the coherency of memory access and negatively impacts memory throughput. I/O streaming accesses multiple data points in each data atom. For I/O streaming, we unroll the vector component loop and access the data sequentially in memory. Cache lines are consumed in their entirety and the necessary coefficients are loaded and used only once. Iterating in the appropriate order scans data sequentially and eliminates random memory access.

3.2.3 Distributed Evaluation

Partial-sums evaluation makes it possible to evaluate decomposable kernel computation queries across nodes of the database cluster. When kernels span multiple database nodes, each of the nodes computes the partial-sum for the data points that it stores and returns it to the mediator. The mediator assembles the sums to complete the computation and returns results.

We use distributed evaluation of queries to eliminate replication among database nodes. The first version of our database was constrained to evaluating each query on a single database node (Section 2.3). Direct evaluation requires all of the data to be available and we wanted to avoid the complexity of inter-node queries. As a consequence, we replicated a kernel half width of data around every data partition. Queries near the boundary of the partition access data in the replicated half width within the interpolation kernel. This resulted in a 42% storage overhead [10]. Using partial-sums, we eliminate this overhead and reclaim the storage space. Our evaluation shows that distributed evaluation incurs little overhead.

3.3 Experimental Evaluation

We evaluate I/O streaming query evaluation by partial-sums using micro-benchmark workloads characteristic of user patterns and on user query traces from the JHTDB. Micro-benchmarks isolate the performance benefits by query type, data sharing, and

data access pattern. User query traces show that I/O streaming realizes an order of magnitude performance increase in practice.

Our evaluation compares **I/O Streaming** against several other evaluation techniques that allow us to isolate the important performance factors. These include:

- **Direct:** The direct method of evaluation that processes queries in arrival order and executes a *SELECT* query for each target point in order to retrieve all of the data atoms that cover its kernel of computation. The interpolation computation consists of nested loops that evaluate one component of a vector field after another (e.g. V_x first then V_y then V_z).

- **Sorting:** An improvement on Direct that also executes a *SELECT* query for each target point, but sorts the target points in Morton z-curve order before processing them. We sort the input in Morton z-curve order since the data atoms are organized in this order on disk and we expect the number of disk seeks to be reduced when reading them in this order. For the interpolation computation, we implement the optimizations described in Section 3.2: iterating in the correct order and evaluating the components of a vector field at the same time. This reduces random memory access and improves the cache locality of the computation.

- **Join/Order By:** A direct method that was redesigned to make use of a join. This eliminates the overhead of executing multiple queries and the database

query execution engine can take advantage of efficient read-ahead and prefetching techniques. The method uses an *ORDER BY* clause on the sequence id of the input queries in order to ensure that all of the data for a query have been read-in before performing the Lagrange interpolation. The interpolation computation is performed in the same fashion as for the Sorting method.

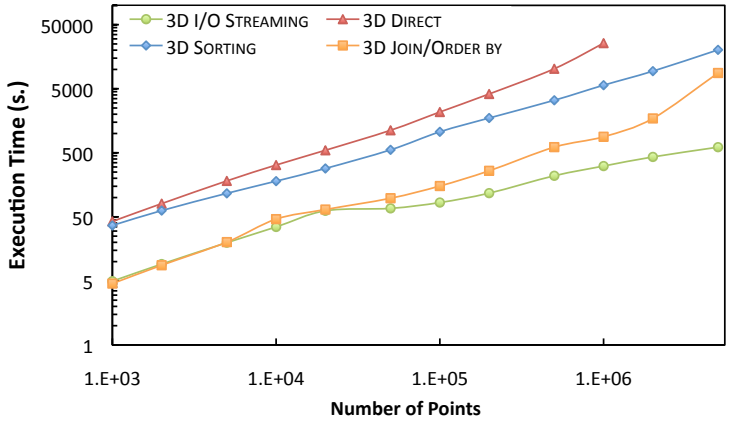
All methods compute 8^{th} order Lagrange interpolation. The Join/Order By strategy executes a single query and takes advantage of database read-ahead and prefetching. Join/Order by is a substantial improvement over the query-at-a-time evaluation of Direct and Sorting. However, Join/Order By does not benefit from data sharing and its performance degrades with respect to I/O streaming for queries that are large or dense in space.

Experiments use a dedicated version of the MHD database cluster that stores ∼300 time-steps of the velocity fields of the MHD DNS (out of a total of 1024 time-steps for the entire simulation). The data are evenly partitioned across two databases based on splitting the z dimension. Database nodes are 2.33 GHz dual quad-core Windows 2003 servers with SQL Server 2008 and 8GB of memory. Data tables are striped across seven disks on each of the nodes. We do not perform experiments on the production version of the JHTDB as our results would be affected by queries run by the users and would negatively impact scientists' workloads.

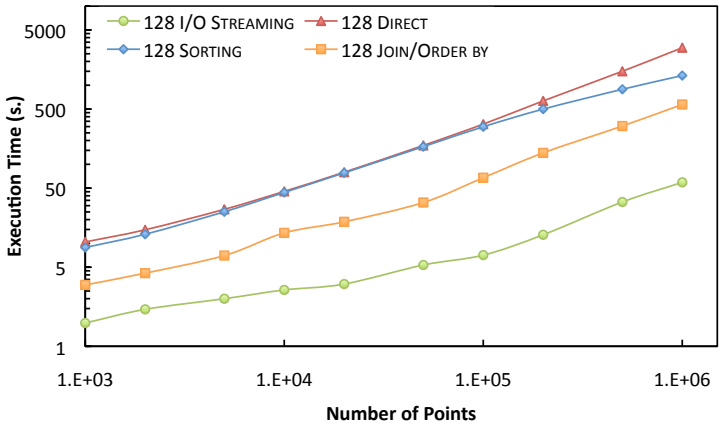
3.3.1 Micro-benchmarks

To define micro-benchmarks, we analyzed the query logs and selected two query patterns that are common among users, but have differing data requirements, sharing, and spatial extent. The **3D** workload selects points distributed randomly throughout the entire cubic volume of a specific time-step. Users employ this query pattern to generate unbiased global statistics. The **128** workload selects randomly distributed points within a randomly chosen sub-volume of size 128^3. Random points within sub-volumes are useful for particle tracking along field lines in a region of interest and creating animations of 3-d turbulence within a memory/space budget. We vary the number of points in a batch query between 1000 and 5 million in order to examine I/O scalability and data sharing. Batch sizes of 100,000 or more are typical of user workload.

Our principal finding is that I/O streaming improves the performance of direct evaluation by an order of magnitude over point query evaluation techniques (Direct and Sorting). Figure 3.3 compares the execution time of all methods, displayed in log scale. Sorting improves performance by up to a factor of two, because it generates I/O patterns that are more sequential. However, Sorting evaluates each query one at a time and reads the same data multiple times when it is accessed by multiple kernels, incurring cache misses. I/O streaming reads each element only once. It never takes a cache miss and accesses data more coherently, in storage and memory order. This results in a further ten times performance gain.



(a) 3D



(b) 128

Figure 3.3: Execution time for randomly distributed points in the entire 1024^3 space (**3D**) and in a 128^3 subset of the entire space (**128**).

For queries without data sharing, the performance benefits come from evaluating queries as joins. Join/Order By requests the data atoms needed by each target point and executes the query as a single join. This is much more efficient than the multiple selection queries used by query-at-a-time methods. Join/Order By execution time tracks that of I/O streaming for smaller queries in the 3D workload. Because 3D randomizes target points over the entire 1024^3 volume, there is essentially no data

sharing for fewer than 10,000 queries. The sort induced by the *ORDER BY* clause also does not have a significant impact in those cases. Joins allow the database query plan generator to pick the most efficient plan for the execution of the query. For example, for batches of up to 10,000 target points a nested loops join with unordered prefetch is executed. For batches of 20,000 target points, a sort merge join is executed. For 50,000 and more points, the database chooses a hash match join. At this point, prefetching and read-ahead become very effective and we see the execution time increasing at a lower rate. Beyond 50,000 points, the physical I/O remains more or less constant, because read-ahead prefetches the entire stored data volume of a time-step.

For larger queries and queries with data sharing, I/O streaming benefits from more effective cache usage. Each database atom is accessed only once even if it is needed by multiple queries. This accounts for the differences between I/O streaming and Join/Order By. If an atom is needed by more than one query it will be accessed more than once due to the *ORDER BY* clause in the *JOIN* statement, and if the atom was evicted from the database cache it will have to be read from disk again. In the 128 workload, even a small number of queries share data, because the volume is restricted. Therefore, the negative effects of accessing the same data multiple times degrades performance for as few as 1000 point queries.

A closer look at I/O shows that the strictly sequential access pattern of I/O streaming makes prefetching effective, which helps account for the large performance improvement over direct evaluation. Figure 3.4 shows the aggregate I/O statistics

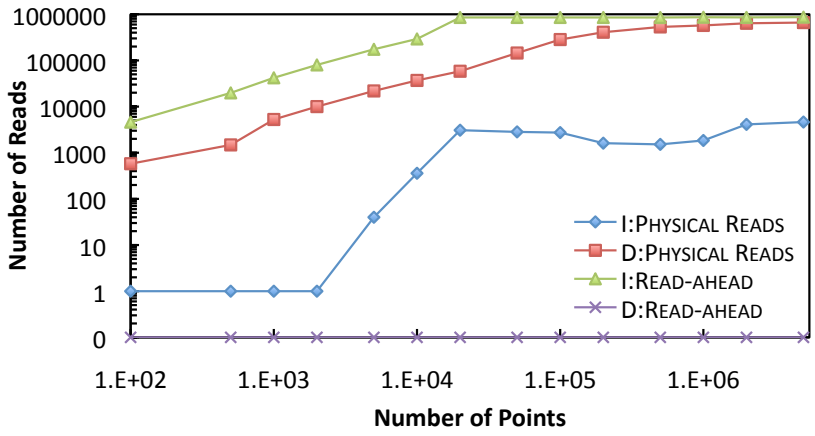
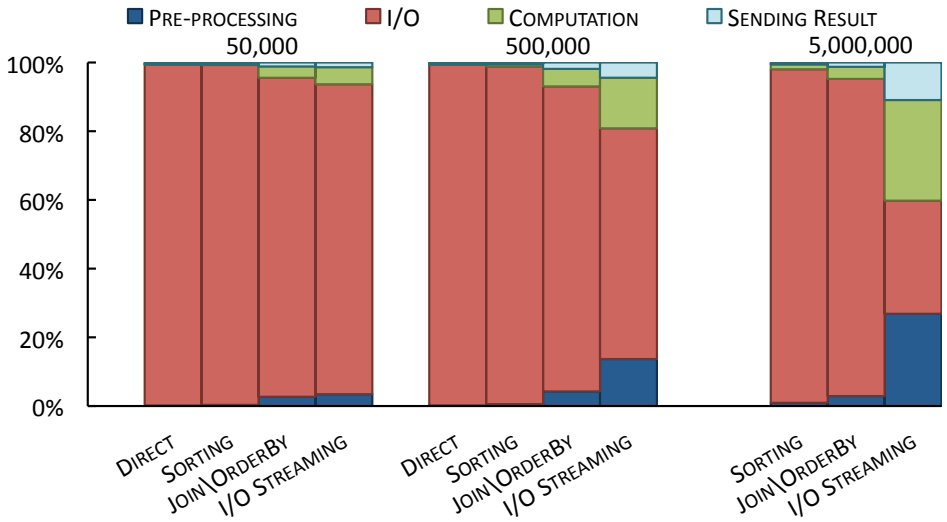


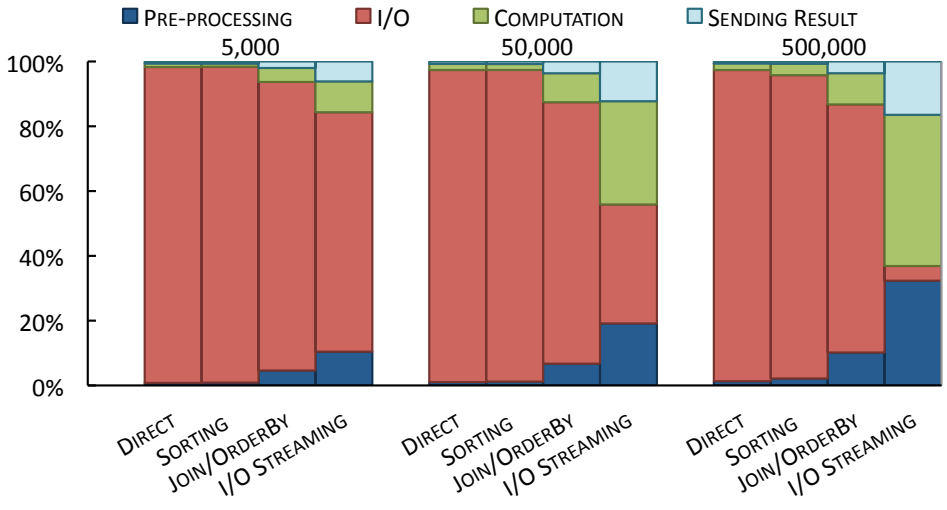
Figure 3.4: Physical and read-ahead reads of I/O streaming (I) and the direct evaluation (D).

generated by SQL Server 2008 for the 3D queries. Executing a single join query results in substantially fewer physical reads. I/O streaming performs up to 500 times fewer physical reads when compared to the direct evaluation (blue and red lines in Figure 3.4). Instead, the database performs efficient read-ahead, which populates the cache with data. Database reads to prefetched data are logical (cache hits) and do not generate physical I/O. On the other hand the direct methods evaluate queries one at a time and do not take advantage of read-ahead. The read-ahead of the direct evaluation is effectively 0 for all queries as shown in Figure 3.4 (purple line). All database reads result in physical I/O, which explains the poor performance.

Decomposing queries into their component costs reveals that I/O streaming alleviates the I/O bottleneck for large data-intensive turbulence queries and computation of the interpolation function emerges as the most costly operation. Figure 3.5 shows the breakdown of execution time into query pre-processing, I/O, computation of the interpolation function, and transmission of query results for the 3D and 128 work-



(a) 3D



(b) 128

Figure 3.5: Breakdown of the execution time for randomly distributed points in the entire 1024^3 space (3D) and a 128^3 subset of the entire space (128). For more than 10^6 points Direct was not executed because the time exceeds reasonable bounds.

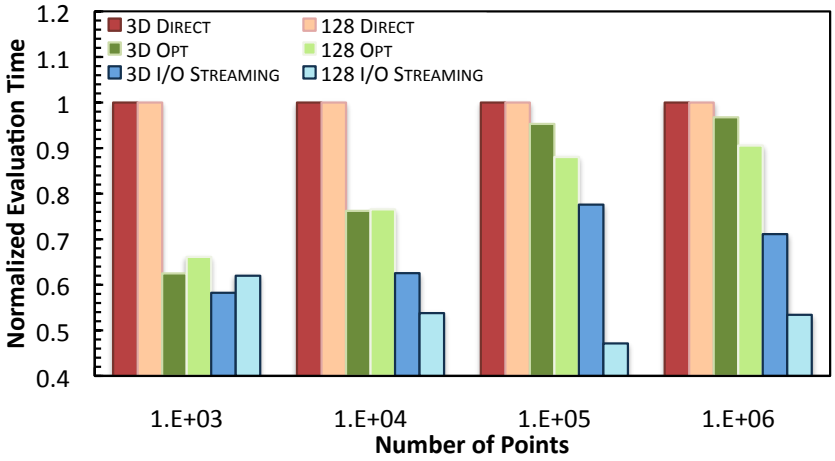


Figure 3.6: Computation time for interpolation comparing the direct method (Direct), loop unrolling and precomputing coefficients optimizations (Opt), and optimization plus I/O streaming data sharing (I/O Streaming). Computation times are normalized to Direct.

loads respectively. For 3D, the computation and I/O costs are roughly the same at 5 million query points. When target points become dense enough, the query reads the entire data space (12 GB for a time-step) and I/O costs stop increasing. The 128 workload accesses a smaller data region (25 MBs) and computation dominates for much fewer than 1M target points. I/O streaming incurs moderate pre-processing costs to determine the Morton z-indexes of the data atoms required by each query and generate the hash table that maintains them along with the partial-sums as described in Section 3.2.1.

As computation becomes a bottleneck, efficient interpolation and memory coherency become important. Rearranging and unrolling loops to reorder memory requests so that they are sequential and pre-computing Lagrange coefficients [20] cuts computation costs by 15% on average and as much as 35% (Figure 3.6). Moreover,

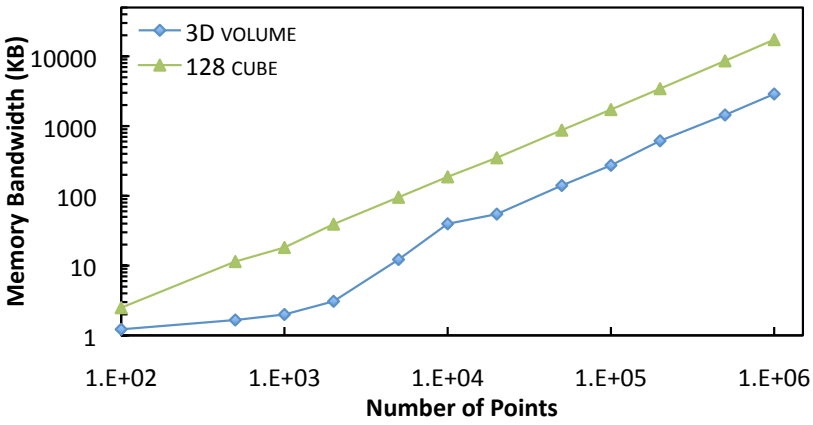


Figure 3.7: Memory bandwidth of an I/O streaming evaluation of Lagrange interpolation.

evaluating by partial-sums effectively exploits the data sharing opportunities that are available. It allows us to iterate over a data atom that is read into memory directly without having to copy data from it into a separate buffer. Because we do this for all queries that need data from the particular data atom, the computation costs are reduced up to a further 40%. We chose the size of a data atom to be 782B so that it fits within L1 cache and all partial-sums using that data are evaluated as L1 cache hits.

The benefits of I/O streaming come at the cost of memory consumption to store partial results. I/O streaming requires space for the results to be allocated when the I/O stream encounters the first partial-sum and retained until query completion when the stream reaches the last partial-sum. Queries with interpolation kernels that span Morton z-order partition boundaries may have to wait quite awhile.

We define the maximum memory needed by active queries as the computation's memory bandwidth (Figure 3.7). Even the largest 3D computations use a negligible

amount of memory: 3MB for 1 million query points. Points are spread out across the entire data space and only a small set of queries are active at any one time. The 128 workload has higher memory bandwidth, because the density of query points in small data regions means that a large fraction of queries can be active concurrently. However, the 10+ MBs needed for 1 million query points still fits easily within cache. Each query uses ∼150 bytes, including 96 bytes of cached coefficients for the Lagrange interpolation. To save space, the coefficients could be recalculated for each partial-sum, but that is unnecessary given the small memory consumption in practice.

3.3.2 User Workload

We perform a similar evaluation on workload gathered from the usage log of the JHTDB. The workload was chosen from a representative 10 day period beginning 05/21/2009. Figure 3.8 compares the execution time for queries of different sizes. The results are similar to the micro-benchmarks with I/O streaming performing up-to 8 times better than Join/Order By. The bump in the execution time for the batches of 900 and 1000 queries is due to the fact that target points in these queries were distributed randomly in the entire space, whereas for most of the other queries the target points were densely clustered together.

Figure 3.9 shows the amount of data sharing as the average number of queries per atom for the user workload. The correlation between available data sharing and

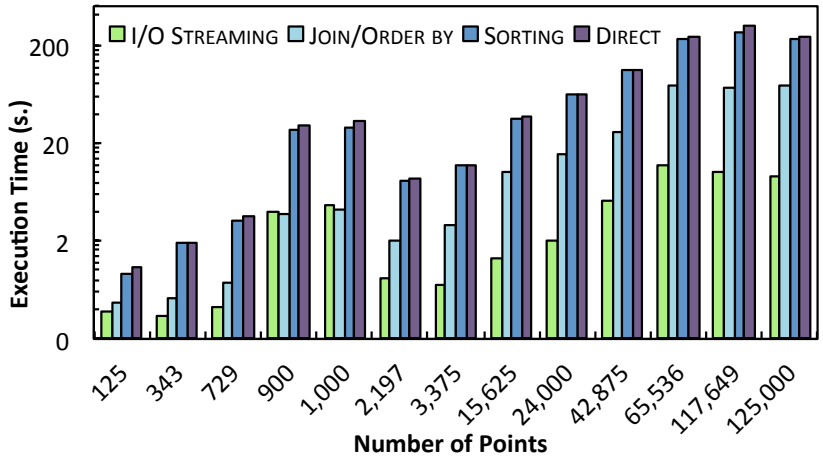


Figure 3.8: Execution time for queries derived from the usage log of the JHTDB.

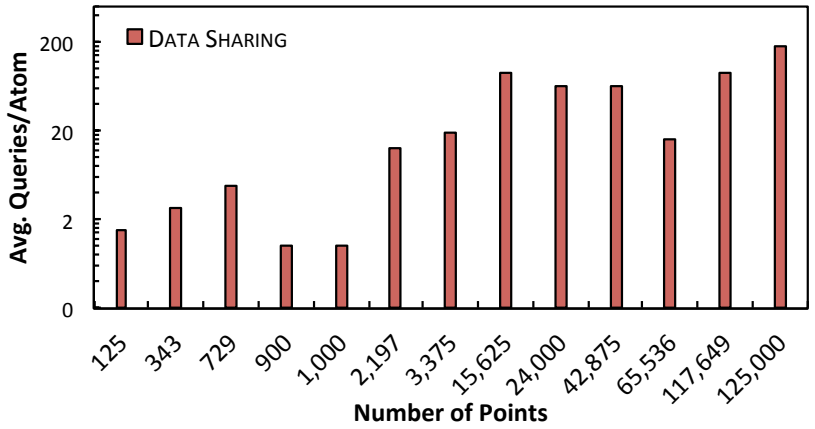


Figure 3.9: Amount of available data sharing for batch queries derived from the usage log of the JHTDB (computed as average number of queries per atom for each batch query).

execution time is evident as less data sharing leads to larger execution times and more data sharing to smaller execution times.

Figure 3.10 shows the time to perform the Lagrange interpolation computation on the user workload. The result is consistent with the micro-benchmark evaluation. The optimized version of the computation reduces the time by ∼15% and the I/O streaming reduces this by a further 40% on average. This result is more closely aligned

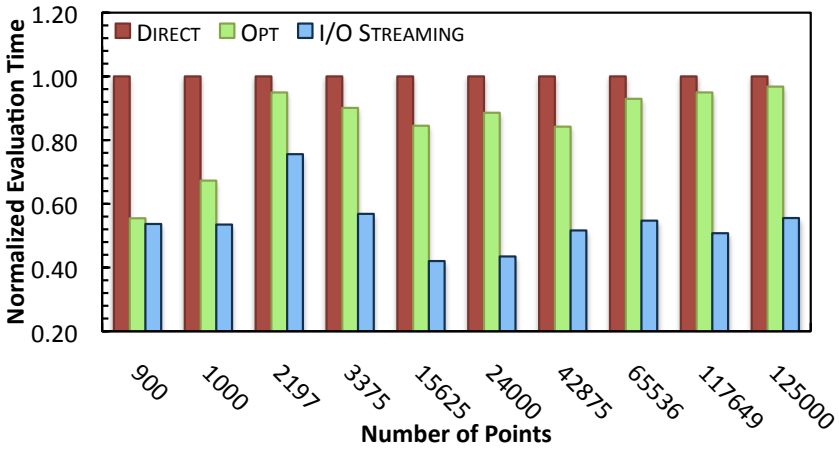


Figure 3.10: Execution time compared for the direct method (Direct), loop unrolling and precomputing coefficients optimizations (Opt), and optimizations plus I/O streaming (I/O Streaming) for queries derived from the usage log of the JHTDB. Computation times are normalized to Direct.

with the results presented for the 128 workload as opposed to the 3D workload. This is due to the fact that most of the user queries were densely clustered in a small region of space, as is the case in the 128 workload.

3.3.3 Distributed Evaluation

Partial-sums computation of Lagrange interpolation eliminates the need to localize each computation to an individual database node, which reclaims the 42% storage overhead of replicating an overlapping data region along partition boundaries. Instead, partial-sums are used to perform distributed evaluation on multiple nodes, evaluating multiple partitions of the Lagrange interpolation kernel on different databases and combining their contributions on the mediator. This incurs some runtime overhead, because the mediator performs additional computation.

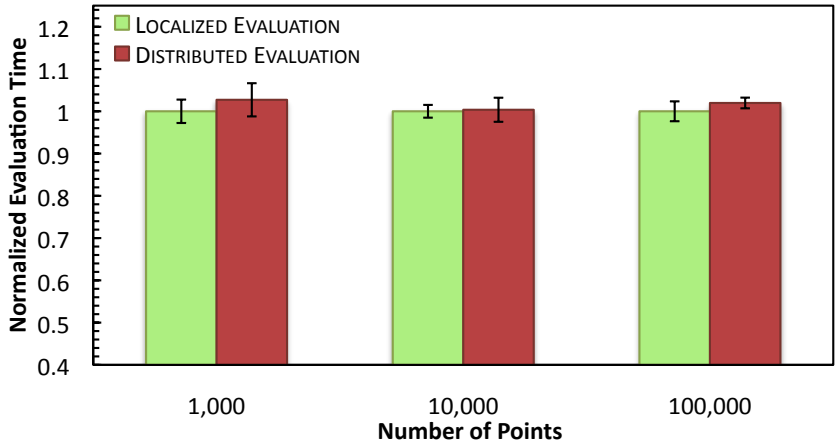


Figure 3.11: Performance of local and distributed computation of partial-sums. (Normalized to local execution time).

A micro-benchmark experiment shows the overhead of distributed evaluation to be about 2%. We configure a cluster of eight virtual database nodes deployed on the two node experimental cluster. We compare two configurations: one that stores an entire time-step on a single node (no distributed evaluation) and the other that divides each time-step into eight partitions by splitting each dimension in two. We then query a 2-d plane across the middle of the time-step on a warm cache, which takes 2% longer on average in the distributed case (Figure 3.11). The warm cache is necessary to reveal the performance difference, otherwise I/O costs dominate. This particular 2-d query has poor distribution performance, because the interpolation kernel of every target point spans 2, 4, or 8 partitions. We conclude that distribution overheads are negligible.

3.4 Related Work

Batch Processing: Data-intensive computing relies on the workload properties of batch computations to realize high-throughput. This is a fundamental tenet of Map/Reduce frameworks [21], which encapsulates computation and data access patterns in the functional abstractions of map and reduce. Shared scans between jobs that access the same files improve the I/O performance of map/reduce [22]. The same principles have been applied in databases [23, 24] that merge queries that share data. All-Pairs [25] and Wavefront [26] use declarative abstractions to computing functions over set combinations and in recurrences respectively. Programs in these abstractions are parallelized automatically and data and computation are placed to minimize I/O. These systems optimize I/O across multiple jobs based on co-scheduling jobs that share data. I/O streaming operates at a finer granularity within a single job, reformulating overlapping kernel queries as a single join and processing data in a strictly sequential fashion.

Paradise [27] uses *query batching* to execute multiple queries that access tape-resident data. Queries are grouped into batches based on the tapes that they access and reordered in order to perform sequential I/O in a pre-execution step. Our I/O streaming evaluation also consumes data in the order in which they arrive and has the additional benefit that it does not impose any restriction on this arrival order. In contrast to *query batching* in the Paradise system, data do not have to be buffered

and are consumed immediately upon retrieval. This lowers the memory consumption significantly and does not pollute caches with buffered data.

Crescando [28] executes multiple queries and updates (operations) by means of a join over sets of operations and a table. We adopt a similar strategy in pre-processing queries into a temporary table and joining the temporary table against a data table. We extend their model by performing the computation in parts because each query point requires multiple data items for completion.

Stream Processing: Scientific applications have been mapped to stream processors that are required to process data sequentially [29, 30]. Yang et al. [30] extract the interdependence of arrays and loops and perform optimizations that include coarse-grained program transformations (loop reordering and fusion, unifying arrays) and fine-grained program transformations (reordering computation and data inside loops). Our streaming approach extends to I/O as well as computation and applies to more complex overlapping kernel functions.

Romein et al. [31] present a software approach to process streaming telescope data on a supercomputer. They reduce the number of memory references during the correlation of signals by keeping correlations in registers and reusing the samples. This exploits the data sharing opportunities that arise during the computation similar to I/O streaming.

Querying Continuous Functions: Extensions to SQL [32] and an algebra for scientific data sets [33] have proposed integrated interpolation, including query opti-

mization. I/O streaming is a promising technique for evaluating such interpolation procedures were the extensions adopted.

MauveDB [34] defines model-based views that represent sparse and irregular raw data as a "uniform grid-based approximation." They also discuss interpolation-based views in which the values at target points are a function of neighbors. Grumbach et al. [35] develop a query language that extends the standard relational framework to include interpolation. FunctionDB [36] defines a query language and algebraic query processor for the creation and querying of function views, interfaces to continuous functions based on regression. None of these works focus on batch queries.

3.5 Discussion

I have presented an I/O streaming technique for the evaluation of data-intensive workloads that consist of decomposable kernel computations, which are used by 95% of queries to the JHTDB. Example functions include differentiation, integration, and filtering. Users submit multiple point queries to be executed in a batch. I/O streaming computes the entire batch in a single, sequential pass over the data by maintaining a partial-sum of the result for each point. The partial-sum of a query is updated whenever the I/O stream produces data within the point's kernel. When compared with direct methods of computation of interpolation, I/O streaming performs an order of magnitude faster.

The single I/O streaming pass over the data improves cache locality and reuse at all levels of the memory hierarchy. The method takes advantage of the efficient execution of joins in modern database systems. It makes full use of read-ahead and prefetching and accesses data sequentially in memory and on disk. I/O streaming exploits the data sharing opportunities that arise during the evaluation of multiple queries and achieves execution times that increase at lower rates for large batches of points and batches whose points are densely clustered.

The focus of this work has been the optimization of a single job containing a batch of target points. We require the single job so that we can represent the batch as a join and preprocess the kernels to order the data access and identify data sharing among kernels. However, data sharing also exists among multiple unrelated jobs that can be leveraged to reduce I/O and improve throughput for map/reduce systems [22], scientific databases [23], and even specifically in the Turbulence Database Cluster [24]. Turbulence workloads often have multiple jobs that analyze overlapping sets of time-steps and could share I/O, i.e. read the data once for all jobs. The job-aware workload scheduler (JAWS) [24] detects this sharing and co-schedules jobs and time-steps. However, JAWS does not manipulate query scheduling and I/O ordering at a fine granularity and realizes much more modest performance benefits than does I/O streaming. We are currently investigating how to integrate partial-sums, I/O streaming, and distributed evaluation into a multi-job scheduling framework, such as JAWS for turbulence data and shared-scans for Map/Reduce [22].

Chapter 4

Data-Intensive Spatial Filtering

In this chapter, I describe a query processing framework for the efficient evaluation of spatial filters on large numerical simulation datasets stored in a data-intensive cluster [37]. Previously, filtering of large numerical simulations stored in scientific databases had been impractical owing to the immense data requirements. Rather, filtering was done during simulation or by loading snapshots into the aggregate memory of an HPC cluster. Our system performs filtering within the database and supports large filter widths. The system makes use of two complementary methods of execution: ***I/O streaming*** computes a batch filter query in a single sequential pass using incremental evaluation of decomposable kernels (Section 3), ***summed-volumes*** generates an intermediate data set and evaluates each filtered value by accessing only eight points in this dataset. We dynamically choose between these methods depending upon workload characteristics. The system allows us to perform filters against large

data sets with little overhead: query performance scales with the cluster's aggregate I/O throughput.

4.1 Motivation

Data-intensive architectures have emerged as attractive platforms for storing and managing large datasets generated from numerical simulations. These systems achieve high aggregate throughput based on I/O and network bandwidth [16]. The JHTDB is built on top of such an architecture and it executes a variety of data-intensive computations on the nodes of the database cluster, including spatial and temporal interpolation, spatial differentiation, and fluid particle tracking (see Section 2.4.2). However, filtering operations had previously remained out of reach, owing to their immense data requirements. The availability of such functionality has greatly enhanced the utility of the datasets stored in the JHTDB.

Many studies of scale interactions in computational turbulence require spatial filtering of the vector and scalar fields resulting from simulations [1]. Filtering or coarse-graining consists of computing a convolution in real space of a filter kernel and a vector or scalar field (or multiplication in Fourier space). The operation is fundamental to analysis in disciplines as diverse as signal processing, geostatistics, and computer graphics. For large data sets, these operations are performed typically on individual time-steps (or snapshots) stored in the aggregate memory of an HPC

cluster. In order to filter world-class simulation data outside of HPC environments, techniques need to be developed that operate on data sets accessed from disk drives. The goal of our work is to make spatial filters efficient to evaluate in real-space on data-intensive clusters.

The difficulty with implementing filtering in real space stems from its massive data requirements. The amount of data that needs to be accessed scales with both the number of locations at which the filtered values are evaluated and with the width of the filter kernel, which can be a substantial fraction of the resolution of the entire simulation. A naive approach that evaluates each value independently reads the same data multiple times when the filter kernels of multiple points overlap. Computing in Fourier space is impractical as this requires the computation of the Fourier transform of an entire snapshot.

My collaborators and I provide a query processing framework that performs filtering based on purely sequential reads and writes in which I/O bounds performance. The framework incorporates two algorithms for the evaluation of filtering workloads. For sparse workloads on smaller kernels, we use the I/O streaming batch query processor [19] (Section 3). For larger, dense workloads, we introduce a two-pass *summed-volumes* algorithm that dynamically builds an intermediate data set based on summing values of each variable to be filtered and evaluates the filtered values in a subsequent pass over the intermediate data. This process was inspired by the use of summed-area tables for texture mapping in computer graphics [38], extending

it to operate sequentially over dynamically generated three-dimensional data. Both techniques are *data-driven* in that they exploit data sharing among kernels by decomposing computations into partial-sums. For each batch query, we choose between these techniques depending upon workload characteristics, such as the total amount of data to be read and the number of target locations to be filtered. The I/O streaming technique supports general filter functions, while the summed-volumes technique is tailored to the box filter function.

The summed-volumes method generates the intermediate dataset of the summed values of the variables to be filtered on-demand, rather than using a precomputed and stored version of this data. While it is possible to precompute and store such a dataset, each precomputed field is as large as the original data and would require tens of terabytes of additional storage. This in turn limits the number of variables that can be filtered. Different scientific analyses require filtering multiple variables and non-linear combinations of variables. On-demand computation evaluates any such combination of variables at runtime from the original (unfiltered) data.

The query processing framework evaluates batch filter queries for computational turbulence 5 to 40 times faster than a naive method of execution. It scales well to multiple nodes in a scientific database cluster and makes effective use of its aggregate I/O throughput. We show results for up to 8 database nodes with 4 virtual servers per node for a total of 32 virtual servers with 1.0 GB/s aggregate I/O read rate.

4.2 Filtering in Computational Turbulence

We describe the use of spatial filtering in computational turbulence in order to motivate how this function enhances the utility of the JHTDB. Specifically, we characterize how filtered fields from our databases will improve sub-grid modeling in large-eddy simulations.

A common approach to studying turbulent flows conducts numerical simulations of the Navier-Stokes equations. Such *direct numerical simulations* (DNS) resolve all time and length scales of the solution. The limitation of DNS is that interesting, real-world turbulent flows are extremely expensive and, in most cases, impractical to compute, because of the amount of computation needed to resolve the smallest scales at high resolution.

In contrast, Large Eddy Simulations (LES) compute only the large scales and model the smallest scales in order to reduce computational requirements. Thus, LES can be used to model much more complex flows. However, the development of realistic models for the small scale motions remains an open research problem.

In LES, the Navier-Stokes equations are transformed by means of low-pass filtering and solutions yield a filtered velocity field. Separating the small and large scales of the motion is done by convolving the velocity filed with a kernel, $G_\Delta(\mathbf{r})$ [39] in which Δ is the length-scale, down to which the fluid motions are resolved. The filtering

operation (denoted by an over-line) is then given by

$$\overline{\mathbf{u}}(\mathbf{p},t) = \int G(\mathbf{r})\mathbf{u}(\mathbf{p}-\mathbf{r},t)d\mathbf{r}, \tag{4.1}$$

in which the integration may be over the entire domain of the flow, depending on the spatial support of $G(\mathbf{r})$.

In order to evaluate the models used in LES, one can compare the simulation results with available experimental data or data from a DNS. To enable meaningful comparisons [40], the latter must be filtered or coarse-grained to a resolution comparable to the LES. Another important method to evaluate sub-grid scale models in LES is to study field variables that arise from filtering the Navier-Stokes equations leading to LES. When filtering the nonlinear advection term, one obtains the so-called sub-grid stress tensor, defined according to:

$$\tau_{ij} = \overline{u_i u_j} - \overline{u}_i \overline{u}_j, \tag{4.2}$$

where u_i, u_j are the components of the velocity field. This tensor represents the momentum fluxes associated with the small-scale turbulent motions that are not explicitly resolved in LES, but that must be included in the evolution equation of the large-scales in LES. The fundamental "closure problem" in turbulence [41] is to express τ_{ij} in terms of the filtered large-scale velocity field $\overline{u}_i$ (sub-grid scale modeling). The ability to measure the "exact" stress field according to its definition (Equation

4.2) from a full solution of the Navier-Stokes equations is important to improve the quality and accuracy of sub-grid scale models.

There are a variety of different filter functions that can be used, including the spectrally sharp filter, the Gaussian filter, and the box filter [1]. The I/O streaming method (Section 3) can be applied to any filter function, whereas the alternative summed-volumes method applies only to a filter function with uniform weights in the convolution kernel, i.e. the box filter. Many studies use the box filter because it has local support and is easily computed by the average of the values in the region around the target location: the points $\mathbf{p}$, where $(\mathbf{p}_0 - \frac{1}{2}\Delta) \leq \mathbf{p} \leq (\mathbf{p}_0 + \frac{1}{2}\Delta)$ given a target location $\mathbf{p}_0$.

To implement filtering with I/O streaming, the coefficients l_i in Equaltion 3.1 define the filter function or convolution to be applied. For example, for a box filter all of the coefficients are equal to the inverse of the volume of the region defined by the filter width as in that case the filtered value is equal to the average of the values in this region.

4.3 Computing Summed-Volumes

We describe the summed-volumes technique for evaluating box-filters on the largest batch queries: those that filter many points at large kernel widths. These queries also capture some of the most interesting science, such as creating a high-resolution

smoothed view of a region of interest. The I/O streaming technique is more general, it supports arbitrary filter functions, and is often more efficient. However, I/O streaming exhibits scalability problems when each input data contributes to many kernels. It becomes expensive to compute the partial-sum for each target filter kernel individually. Summed-volumes avoids this problem by sharing partial-sums computations among all kernels at the expense of a two-pass algorithm and the generation of an intermediate dataset. As a result, the summed-volumes technique exhibits stable performance over all parameterizations of kernel sizes and numbers of points.

The summed-volumes algorithm is inspired by the use of summed-area tables for texture mapping [38]. We extend the technique to volumetric data, to support sequential I/O, and to dynamically compute the sums rather than precomputing and storing the dataset.

The first pass of the algorithm determines the bounding data region of the target locations and generates a summed-volumes dataset over that region. Every data point in this intermediate dataset stores the sum of all data values below and to the left of it. The second pass of the algorithm uses this summed-volumes dataset to efficiently compute a box filter for every target location. Figure 4.1 depicts this process for a three-dimensional dataset. We elaborate on the two passes of the algorithm below.

The on-demand computation of a summed-volumes dataset in d dimensions determines the bounding data region of the set of input points, defined by lower left $H_L = (x_{0_L}, x_{1_L}, ..., x_{d-1_L})$ and upper right $H_U = (x_{0_U}, x_{1_U}, ..., x_{d-1_U})$ corners. The

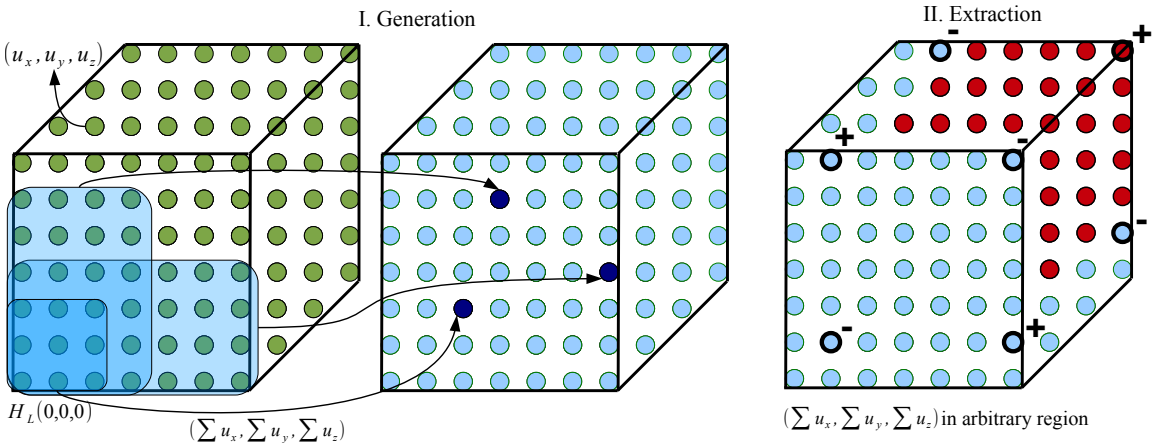


Figure 4.1: Using summed-volumes to compute the sum of grid values over an arbitrary region of interest. Every point in the summed-volumes dataset (middle) stores the sum of all values between the lower left corner $H_L(0,0,0)$ of the grid and that point in the original data set (left). The image on the right depicts the extraction process from the generated summed-volumes dataset. The filtered value in an arbitrary region (shown in red) is extracted by subtracting the sums outside of this region from the value at its top right corner. This process looks at eight data points.

coordinate x_{i_L} is given by $min(x_i) - \frac{1}{2}\Delta$, where the minimum is taken over the x_i-th coordinate of all of the input points. The coordinate x_{i_U} is given by $max(x_i) + \frac{1}{2}\Delta$, where the maximum is again taken over the x_i-th coordinate of all of the input points. This region defines the total amount of data that has to be retrieved and processed.

In the case of the JHTDB, the set of database atoms that cover this region are retrieved and processed in Morton z-order. This is the same order in which data are laid out of disk, ensuring that disk I/Os are performed to increasing offsets, which is as or more efficient than a single sequential pass.

The summed-volumes dataset is manifested on an d-dimensional grid. The value at every grid point in the summed-volumes dataset is the sum of all values contained in the hyper-rectangle defined by the corresponding point in the original dataset

and the lower left corner of the grid (Figure 4.1). The dynamic generation of the summed-volumes dataset proceeds as follows.

Points in the original dataset are summed in Morton z-order in a single pass. The z-order visits points in a grid in non-decreasing order in each dimension; when accessing a point, the sums of all its lower left adjacent points have already been computed. We add the value of point $(x_0, x_1, ..., x_{d-1})$ to the sums already computed for its lower adjacent points to calculate the sum of all points between that data location and the lower left corner of the grid:

$$s(x_0, x_1, ..., x_{d-1}) = u(x_0, x_1, ..., x_{d-1}) + \sum_{i_0=0}^{1} \sum_{i_1=0}^{1} ... \sum_{i_{d-1}=0}^{1} (-1)^{i_0+i_1+...+i_{d-1}-1} \cdot s(x_0 - i_0, x_1 - i_1, ..., x_{d-1} - i_{d-1}) \quad (4.3)$$

in which u denotes values in the original dataset, s denotes values in the summed-volumes dataset and $s(x_0, x_1, ..., x_{d-1})$ is assumed to be initialized to 0. In total 2^d additive operations and 2^{d-1} lookups are performed per data point.

The next step in the algorithm computes the filtered value from the generated summed-volumes dataset. First, we extract the sum of values in the region defined by the filter kernel. The sum of all values lying inside an arbitrary hyper-rectangle defined by its lower left $H_L = (x_{0_L}, x_{1_L}, ..., x_{d-1_L})$ and upper right $H_U = (x_{0_U}, x_{1_U}, ..., x_{d-1_U})$

corners is given by:

$$\sum_{i_0=0}^{1}\sum_{i_1=0}^{1}...\sum_{i_{d-1}=0}^{1}(-1)^{i_0+i_1+...+i_{d-1}}\cdot s(x_0(i_0),x_1(i_1),...,x_{d-1}(i_{d-1})) \tag{4.4}$$

where the coordinates $x_j(i_j)$, for $j \in (0,1,...,d-1)$, are equal to x_{j_U} if $i_j = 0$ and $x_{j_L} - 1$ if $i_j = 1$. Thus, 2^d points have to be accessed.

Computing a box filter takes the average of the data values in a region centered on the target location. With the above technique we can compute the sum of the data values in the filter kernel associated with a target location $p' = (x'_0, x'_1, ..., x'_{d-1})$ by specifying the corners of the region as $H'_L = (x'_0 - \frac{1}{2}\Delta, x'_1 - \frac{1}{2}\Delta, ..., x'_{d-1} - \frac{1}{2}\Delta)$ and $H'_U = (x'_0 + \frac{1}{2}\Delta, x'_1 + \frac{1}{2}\Delta, ..., x'_{d-1} + \frac{1}{2}\Delta)$. Therefore, given the sum in that region all that one has to do is divide by the region's volume.

In three dimensions, we access eight points in the summed-volumes dataset to compute the sum in a rectangular region. We then multiply by a filter function in order to compute the filtered value. In this case only constant filter functions (e.g. the inverse of the volume) can be used as the sums are precomputed. The rightmost image in Figure 4.1 shows the points that have to be accessed and whether the values are to be added or subtracted in order to compute the sum of the points in red (one of the points accessed is not shown as it is in the back of the grid).

On-demand computation of the data sets allows us to support user-defined filters using complex expressions, such as $\overline{u_i u_j}$ that is needed to evaluate the sub-grid stress

tensor (Equation 4.2), While the intermediate data could be precomputed, this quickly becomes impractical. We have already identified several interesting filtered fields and each precomputed field would be as large as the storage for a variable from which it is derived. For example, the 1024 time-steps of the raw velocity field data alone occupy 12 TBs on disk. Precomputing and storing summed-volumes datasets for each time-step for the 9 quantities needed for the evaluation of the sub-grid stress tensor will require 36 TBs of additional storage.

4.4 Putting It All Together

The I/O streaming and summed-volumes techniques exhibit different performance characteristics and dynamically choosing between them on a per-query basis improves performance dramatically. Summed-volumes works better for workloads in which the filter kernels have substantial overlap. In these cases, both techniques retrieve the same amount of data from disk. However, I/O streaming routes each data point to all of the kernels that need it, performing a different computation for each data point in each kernel. By manifesting an intermediate data set, summed-volumes shares the computation of each data point among all kernels. This reduces computation, but it also restricts summed-volumes to box filters in which each data point contributes equally to each kernel.

Our process for choosing between I/O streaming and summed-volumes relies on

analysis of the scaling properties of these techniques. We then determine how to parameterize this analysis with experimental results to define a dynamic query optimizer. Given an atom size s^3 and kernel size k^3, the average number of data atoms that have to be accessed by a query is given by $(\frac{k+s-1}{s})^3$. Therefore, since the data are partitioned into atoms, for a batch consisting of p queries, the total number of individual data points that need to be accessed can be estimated as $p \cdot (k+s-1)^3$. We define the density of input queries or kernel overlap as the number of queries accessing the same data point. It can be estimated as $\rho = \frac{p \cdot (k+s-1)^3}{x \cdot y \cdot z}$, in which x, y, and z define the dimensions of the bounding data region for the entire batch.

We analyze both the I/O and computation scaling properties of both techniques. I/O streaming retrieves $p \cdot (\frac{k+s-1}{s})^3$ atoms from disk and performs roughly $p \cdot (k+s-1)^3$ operations because the size of each atom is s^3. Summed-volumes retrieves the entire bounding region of data, $\frac{x \cdot y \cdot z}{s^3}$ atoms, generates the intermediate data set using $8 \cdot x \cdot y \cdot z$ operations and extracts the filtered results in $8 \cdot p$ operations for a total of $8 \cdot (x \cdot y \cdot z + p)$ operations.

When kernels have little overlap, density $\rho \leq 1$, I/O streaming accesses less data and performs fewer operations. I/O streaming is always preferred. When kernels overlap, density $\rho > 1$, the best choice depends upon the density and number of points. We assume that I/O streaming and summed-volumes access the same amount of data. The data atoms covering the entire region $(\frac{x \cdot y \cdot z}{s^3})$ provide an upper bound for I/O streaming and an exact figure for summed-volumes. The I/O requirements match

in practice, because I/O streaming's best strategy performs a sequential read of the entire data volume. The techniques differ only in their computational requirements.

We empirically determine the parameters at which the computation of I/O streaming and summed-volumes match. We use summed-volumes if $\frac{p \cdot (k+s-1)^3}{8 \cdot (x \cdot y \cdot z + p)} > 10$ and use I/O streaming otherwise. For most batch queries $p \ll x \cdot y \cdot z$, the fraction $\frac{p \cdot (k+s-1)^3}{8 \cdot (x \cdot y \cdot z + p)}$ is approximately $\frac{\rho}{8}$. The constant 10 reflects the difference between the sequential memory accesses of I/O streaming and the random memory accesses of summed-volumes. Random memory accesses are $\sim$10 times slower. This inequality covers the case when kernels do not overlap as well, choosing I/O streaming at low densities.

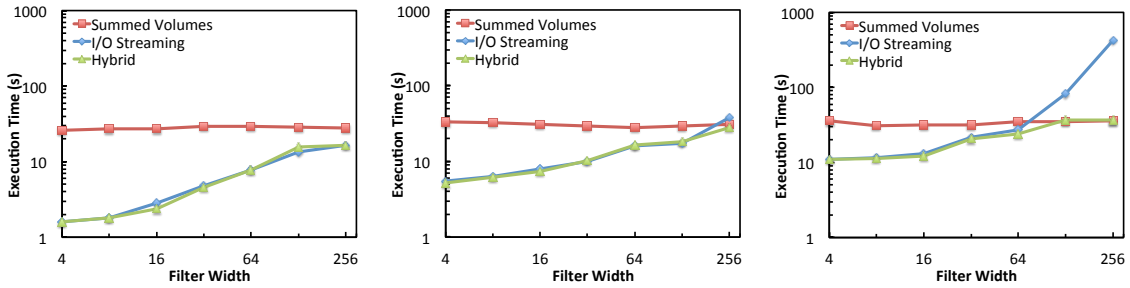
4.5 Experimental Results

We evaluate the performance of the I/O streaming and summed-volumes methods for batch spatial-filtering queries on microbenchmarks and anticipated usage pattern workloads derived from scientists' initial queries. We also compare performance with direct execution of a set of queries that create coarse-grain representations of the entire space: a query type that will be used frequently in practice. The benchmarks show that direct evaluation of individual queries is impractical, while using the developed query processing framework results in 5 to 40 times improvement in the execution time of queries. The query processing framework is also able to effectively determine which method to utilize for all workloads.

The experiments were run through a development Web-server mediator that connects to the production nodes of the JHTDB. The data are evenly partitioned across either 4 or 8 nodes according to spatial regions in the Morton z-order. Database nodes are 2.33 GHz dual quad-core Windows 2008 servers with SQL Server 2008 and 24 GB of memory. Each node is connected to two MD1000 SAS disk boxes that contain a total of 30 750 GB, 7,200 rpm SATA disks. We utilize 4 virtual servers per database node (unless stated otherwise) in order to make use of the multicore parallelism available on each node. Data tables are partitioned across 4 logical data volumes on each of the nodes. The measured I/O read rate through SQL Server Management Studio for our queries is 130 MB/s per node, for a total aggregate I/O read rate of 1.0 GB/s across 8 nodes. All experiments were run with a cold cache.

Figure 4.2 shows the execution time of three batch queries as a function of the filter widths for 1024, 10,240 and 102,400 locations randomly distributed in the entire 1024^3 volume. The query calculates the filtered value of the vector field at each individual location. The figure compares I/O streaming, summed-volumes, and the hybrid method that uses the query processing framework of Section 4.4 to choose between them on a per query basis.

The execution time of summed-volumes remains more or less constant for all three batches and at all of the different filter widths. Summed-volumes manifests the same intermediate data set in all cases. The amount of data that must be read and processed depends only on the bounding data region of the entire batch. Since the



(a) 1024 points (b) 10,240 points (c) 102,400 points

Figure 4.2: Execution time for randomly distributed points in the entire 1024^3 space with varying filter widths.

points are distributed randomly in the entire volume, the entire data volume is read and a summed-volumes data set of an entire time-step is created. Summed-volumes filters an entire 1024^3 timestep in just over 30 seconds. I/O streaming typically outperforms summed-volumes because it performs I/O only to the data atoms needed by the filtering kernels. With fewer points and narrower kernels, this can be much less than an entire timestep as indicated by the results.

However, the performance of I/O streaming degrades for larger numbers of points and wider kernels. At some point, I/O streaming retrieves the entire volume of data. Beyond this point, I/O remains constant and the execution time begins to be dominated by computation, which scales as $p \cdot k^3$ for I/O streaming, in which p is the number of positions and k the filter width. At the largest filter widths in our experiment, I/O streaming can be more than ten times worse than summed-volumes. The query processing framework uses the workload characteristics to effectively decide which method to deploy. The hybrid approach tracks the best performance of either method.

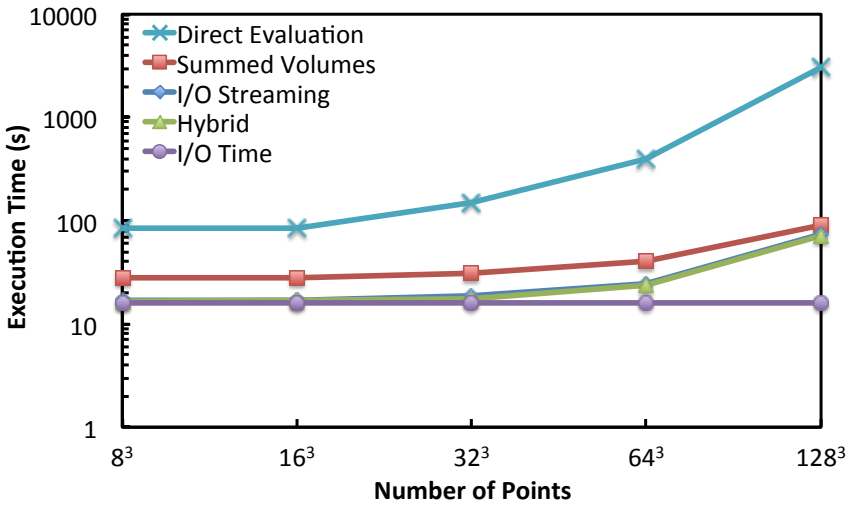


Figure 4.3: Execution time for coarse graining queries covering the entire data volume at different density.

An application of spatial filtering for large scientific datasets is the generation of lower resolution dataset or coarse-graining of the high-resolution data. This is done by filtering the data at locations placed on a cubic lattice, where the points on the lattice are placed so as to cover the entire data volume. The width of the filter used in this case is a function of the separation between the points on the lattice, r. We use $2 \cdot r$ as the filter width. The execution times of queries of this type are presented in Figure 4.3. Since the entire data volume is coarse-grained, all of the data for a time-step is retrieved. The execution time of the I/O streaming method, summed-volumes method and the hybrid method are compared alongside the time required to perform just the I/O. The execution time of a direct implementation of spatial filtering is also presented. In this case the individual queries forming a batch are executed one at a time by retrieving the data necessary for each query and evaluating the filter before moving on to the next.

As the number of locations in the cubic lattice increases the execution time of the direct evaluation of individual queries goes from minutes to hours. The execution time of I/O streaming on the other hand tracks the I/O time for batch queries with a small to medium number of target locations. The method is able to achieve over 1.0 GB/s aggregate read rate in those cases. This makes it five times faster than direct evaluation for the fewest number of points and more than 40 times faster for the largest set of points. Because the query density is small for all workloads of this type, we would expect the execution time to remain close to the I/O time in all cases. The processing that happens on the Web server and the fact that its resources are shared among other development deployments is the reason for the increase seen at a large number of target locations. Because we measure the execution time at the client as the time to submit a batch query and obtain the results, this includes the transfer of target locations to the Web server and the distribution of queries to each of the database nodes. For coarse-graining fields, I/O streaming always outperforms summed-volumes, because as the number of points increases, the filter width decreases so that the kernel overlap of queries remains constant.

Another application of spatial filtering generates a smoothed view of a region of interest. We present timing results for queries of this type in Figure 4.4. The locations where a filtered value is desired are placed on a cubic lattice of size 64^3 and are separated by $4 \cdot dx$, in which dx is the grid resolution of the simulation. At the point where the execution times of the two methods crossover (filter width 16), there

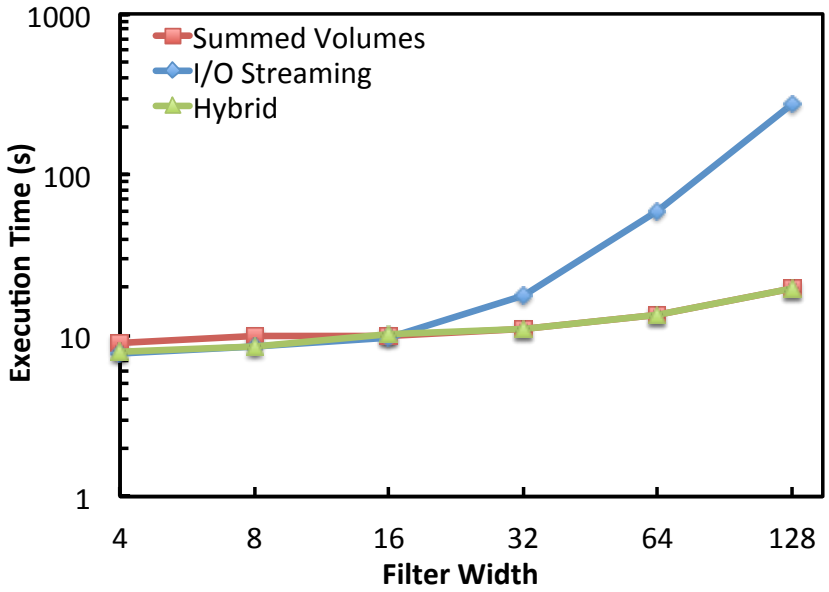


Figure 4.4: Execution times of queries requesting the filtered value of the velocity field on a cubic lattice of 64^3 points.

are already ∼180 queries accessing every data point and this number increases to ∼11,600 for filter width 128. The summed-volumes method clearly performs better for the queries that perform significant smoothing and use large filter widths. The query processing framework detects this for this query type as well and chooses the summed-volumes method for the evaluation of the large filter width queries. It utilizes the I/O streaming method otherwise.

As discussed in Section 4.2, there is interest in computing complex filter quantities of non-linear combinations of simulation parameters, such as the sub-grid stress tensor. In Figure 4.5 we show the execution times of sub-grid stress tensor queries (labeled "SGS") for the same distribution of target locations as in the experiment discussed above (Figure 4.4). We compare these execution times with those request-

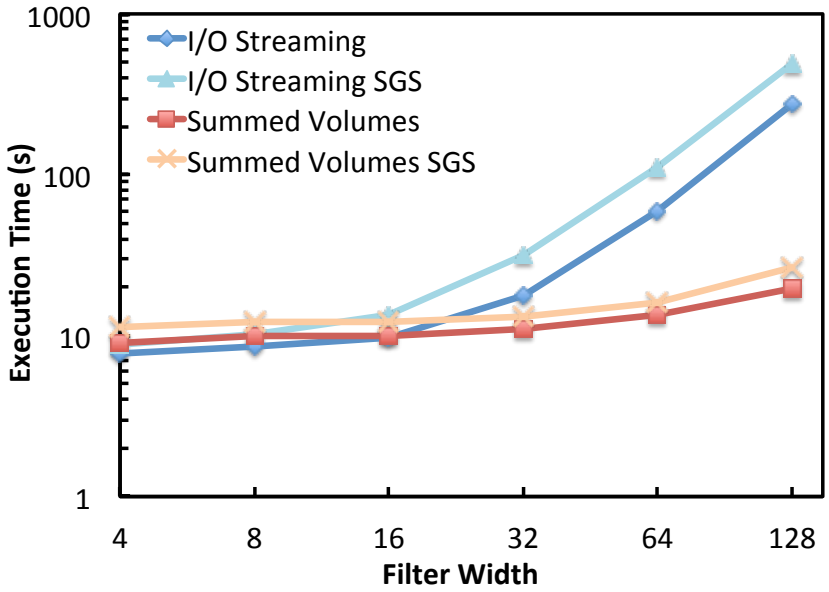


Figure 4.5: Execution times of queries requesting the sub-grid stress tensor (SGS) compared with the execution times of queries requesting just the filtered value of the velocity field on a cubic lattice of 64^3 points.

ing just the filtered velocity components. As we can see in the figure the overhead of maintaining and computing 9 quantities as opposed to 3 is not significant even though intermediate results are three times as large. It is ∼24% on average for the summed-volumes method and no more than 34%. For the I/O streaming method it grows to ∼80%, but only for the queries with large filter widths for which the hybrid technique would choose summed-volumes. Otherwise it is less than 20%.

Our last experiments look at the scale-out of the service and the parallelization of computation using summed-volumes. Figure 4.6 presents the execution times of batch queries consisting of 102,400 points randomly distributed in the entire 1024^3 volume with varying filter widths. These are the same queries as those presented in Figure 4.2(c). The execution times of these queries are evaluated for a database

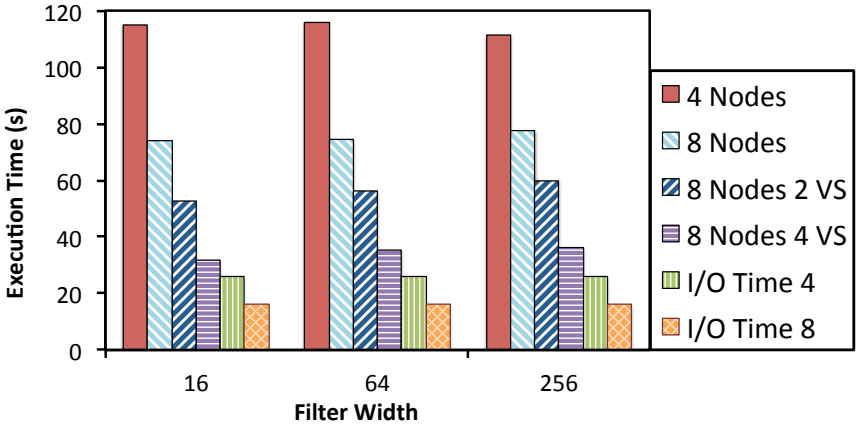


Figure 4.6: Execution time of the summed-volumes method on different server configurations for 102,400 randomly distributed points in the entire 1024^3 space with varying filter widths.

configuration consisting of 4 nodes and another consisting of 8 nodes. For the 8 node configuration, we also present results for a setup that treats each individual node as 2 or 4 virtual servers (labeled "VS"). The data reside in the same database table, but each virtual server only queries the portion that it is assigned. The plot also shows the I/O time for the retrieval of the data only without any additional processing.

Summed-volumes has significant processing requirements, but computations can be parallelized so that I/O consumes about half of overall performance. We parallelize computation by creating virtual servers on each physical node that perform the summed-volumes computations on different cores. Execution time on 8 nodes is roughly 75 seconds compared with 114 seconds on 4 nodes. Adding two and four virtual servers reduces this to 55 seconds and 34 seconds respectively. This realizes an overall speedup of 3.4 and the computation takes just 2.1 times as long as the raw I/O on 8 nodes.

4.6 Related Work

The computation of spatial filters for data-intensive computing has not been extensively studied, likely owing to the fact that computing a filtered value at a single location in the spatial domain of a time series requires accessing a substantial portion of the data of an entire time-step, which can be several GB in size. Additionally, data-intensive architectures and systems have only recently emerged as attractive platforms for the storage of high-resolution numerical simulation datasets. Traditionally filtering has been done during simulation [42] or on large snapshots stored in the distributed memory of an HPC cluster [43].

Filtering has been extensively studied in the field of image processing [44, 45]. The focus of this line of work is the development of new filters and filtering techniques [46] and the parallelization of the computation as the data to be filtered is not large and can fit in the memory of a CPU or a GPU [47]. Our work focuses on out-of-core solutions and external memory algorithms for the filtering of large three-dimensional scientific datasets, storing vector fields.

DataCutter [48] is a middleware infrastructure for subsetting and aggregation of large datasets on archival storage systems. DataCutter uses the term "filter" to refer to subsetting data. This concept is unrelated to the spatial filters in our work, which compute coarse-grained fields by convolution. Such processing tasks are user-defined in DataCutter and we have shown that naive implementations can make their evaluation impractical and render them unusable. The Earth System Grid [49] is a

similar type of infrastructure to DataCutter, which aims to support high-performance, interactive analysis and remote access of simulation data. It makes use of the Climate Data Analysis Tools (CDAT) [50] to perform client-side analyses.

Coarse-graining and data reduction techniques are used for the visualization of large datasets [51]. This is done during a post-processing step [52] or requires parallel rendering algorithms [53] on massively parallel processors. These techniques are not designed with the goal of performing statistical analyses on the data and introduce error that makes them unsuitable for scientific experiments.

Data sieving [54] aggregates noncontiguous accesses to the file system into a few, large contiguous requests. Similarly, the methods that we present aggregate the requests of all queries in a batch. Additionally, with summed-volumes the data is transformed into a summed-volumes dataset for the efficient extraction of filtered quantities.

Database support for array data, such as the *Array Manipulation Language* [55] or SciDB [56], provides ways to define and manipulate arrays. As of now, these systems do not support the optimization of convolutions. I/O streaming and summed-volumes could be incorporated as an execution strategy in such systems. MauveDB [34] introduces model based views for incomplete and sparse data, but does not include a treatment of batch queries with potentially overlapping data requirements.

Push-based database systems such as DataPath [57] take a data-centric approach to query processing. Data are pushed through the memory hierarchy and are shared

between computations. The data are read continuously by means of table scans and pushed through waypoints that perform the required computations. The I/O streaming and summed-volumes techniques also aim to share data and computation among batch queries, but achieve this by means of pre-processing a batch query and retrieving only the necessary data in a single I/O stream.

4.7 Discussion

We have described a query-processing framework for the execution and evaluation of batch queries that apply spatial filters to large numerical simulation datasets. The framework incorporates and extends the I/O streaming method for the evaluation of interpolation and differentiation, described in Section 3. We also introduce a summed-volumes method that resolves the scalability problems of the I/O streaming method for the largest filtering queries. It exhibits stable performance across all workload parameterizations. Summed-volumes dynamically computes an intermediate dataset of summed-volumes for each variable to be filtered and allows for the computation of filters that combine multiple variables, such as sub-grid stress tensor. The query-processing framework dynamically selects the best performing method for each query. The result reduces query processing times by a factor of 5 to 40 when compared with direct evaluation of individual queries.

Summed-volumes computes and stores intermediate data sets in memory and the

memory capacity of the database cluster limits scalability. Computing a spatial filter over the entire data volume of a simulation time-step requires memory equivalent to the size of the time-step (currently 12 GBs per simulated vector field). Each time-step is distributed across the nodes of the cluster and therefore only a fraction of the total amount is retrieved in the main memory of each node. As the time-steps grow in size we expect to be able to allocate new nodes in the cluster, which will allow us to not only store more data but also have additional memory capacity. However, computing more complex filter quantities requires the generation of multiple intermediate fields. For example, computing the sub-grid stress tensor requires twice as much memory as filtering the vector field alone. The intermediate dataset that is stored in memory scales with the number of independent quantities to be filtered and is thus limited by the memory capacity of the cluster.

In our future work, we intend to materialize the intermediate summed-volumes dataset to a temporary table on disk when filter quantities consist of multiple variables that are too large to fit in memory. Creating such a temporary table can serve as a persistent cache and can be placed on an SSD attached to each database node. For queries that hit data in the cache the execution time will be improved substantially as the materialization of the summed-volumes dataset will not be necessary. Each query will be evaluated by looking up only 8 values in the cached dataset. This approach will also require the reevaluation of the density threshold for choosing between I/O streaming and summed-volumes. Materializing the summed-volumes dataset to stable

storage will incur higher I/O costs, but subsequent queries to the same data will not have to recompute the summed-volumes dataset.

We plan to evaluate the effects of parallelizing parts of the execution of the I/O streaming and summed-volumes methods at a finer level. For I/O streaming, when a database atom is routed to multiple queries the computation of partial-sums is independent across the different queries and could be done in parallel. This could improve the scalability of I/O streaming for large numbers of points with overlapping kernels. For summed-volumes, the intermediate summed-volumes dataset could be generated using multiple threads. However, it is important to make sure that at every stage in this process the data below and to the left is already processed. A wavefront access pattern ensures that this is the case and is also amenable to parallelization [26].

The implementation of filtering capabilities increases the utility of archived numerical simulation datasets. Filtering capabilities are invaluable for the study of LES sub-grid scale modeling. They allow for new types of analyses to be performed on the data. Users with modest computing capabilities can execute large filtering queries that are data-intensive, because the evaluation is done on the database cluster transparently to the user.

Chapter 5

Efficient Evaluation of Threshold Queries of Derived Fields

Data-intensive computations that examine the entire data volume of a given time-step of large numerical simulation datasets are impractical to perform locally by the user, taking days or months to iterate over the entire dataset. In this chapter, I describe a method for the efficient evaluation of threshold queries of derived fields for large numerical simulation datasets stored in a cluster of relational databases [58]. The integrated method for the evaluation of threshold queries achieves scalability through data-parallel execution of the computations on the nodes of an analysis database cluster. My collaborators and I have extended the scientific analysis environment with the introduction of an application-aware cache for query results, building on the concept of semantic caching. The cache has little overhead and improves query

performance by over an order of magnitude for queries that hit the cache. Caching the results of threshold queries preserves both the I/O and computational effort used to obtain them. In the case of computational turbulence, this allows scientists to quickly focus on the most intense events and interesting regions in any time-step or the dataset as a whole, which greatly speeds up the rate of scientific exploration and discovery.

5.1 Motivation

Novel data-intensive systems and architectures have been developed in the recent past with the goal of storing and providing fast access to large scientific datasets produced by observation, experimentation or simulation. Examples of such analysis environments include the GrayWulf and Data-Scope clusters [16, 17] at Johns Hopkins. One of their missions is to provide persistent storage and public access to world-class numerical simulation data. These systems differ from the traditional HPC environments in that they aim to achieve high aggregate throughput by balancing computational capabilities with I/O and network bandwidth. The computing systems and services developed on top of these platforms are more than pure storage engines and usually have complex analysis routines built-in, which has largely been driven by the "move the computation to the data" paradigm [59]. These built-in analysis routines are most often not novel themselves. They implement core scientific

functionality for the study of the particular scientific phenomena, which was observed or simulated in the first place. The analysis routines however require novel evaluation strategies and methods for their execution. They have to operate on large array datasets distributed across multiple nodes of a cluster of relational databases. In order to reduce their running times, they have to make efficient use of the cluster resources and incorporate leading data management techniques.

Finding the locations or regions of highest vorticity or those with the largest norms of the velocity or other fields of interest enables new insights in the study of fluid dynamics. Analysis of this kind coupled with the ability to analyze time-series datasets both forward and backward in time has transformed our understanding of turbulence [4]. Furthermore, threshold, top-k queries and similarity search in general are important in many different disciplines. We introduce an efficient evaluation strategy for threshold queries over time-series datasets stored in a cluster of relational databases. Our method evaluates not only threshold queries of the vector or scalar field data stored in the database, but also performs thresholding of *derived* fields.

The main challenge that our approach tackles is that the data-intensive computation of *derived* fields has to be carried out on-demand for extremely large array datasets stored in an analysis cluster environment comprised of multiple database nodes. We focus on the evaluation of threshold queries of fields derived from the data stored in the cluster as these queries are the most interesting scientifically. However, our approach applies to the evaluation of top-k queries, rollup queries and data-

reducing queries in general. The established data management techniques that our approach combines make the approach easy to understand. It can be applied to other scientific analysis environments, which manage large datasets in a database management system. Examples include the Sloan Digital Sky Survey [60], the Millennium Simulation [61] and the Open Connectome Project [62].

Evaluating threshold queries within the database cluster allows scientists with modest computational and networking capability to narrow down on and examine some of the most interesting regions and features in the dataset and focus on the subsequent analysis needed to understand these events. It is impractical to materialize all possible derived fields and store them alongside the raw data due to the large size of the datasets and the limits of available storage. Obtaining the derived field and thresholding locally by the user requires not only the computation of the derived field over the entire data volume of a time-step server-side but also the transfer of a large amount of data over the network, most of which are subsequently discarded. One of our collaborators reported that such a local evaluation of a threshold query over the entire data volume of an individual time-step took over 20 hours. It would take months to iterate over the entire dataset. This highlighted the need for providing the capability through an integrated approach, which performs the evaluation server-side.

Database, operating and file system caches are effective at speeding up access to the large amounts of data stored on disk. However, this might not be sufficient for some applications, because these application-independent caches cannot exploit

dataset-specific structure and application-level information [63]. Moreover, even if the data are available in one or more of these application-independent caches the computation associated with the derived field still needs to be performed for each point on the grid, because results of previous computations are not cached. We will demonstrate that an integrated approach, which computes the derived fields on-demand in a data-parallel manner, performs the evaluation over the entire data volume of a given time-step in a few minutes. Storing the query results in an application-aware semantic cache further reduces the running times to several seconds.

Thresholding allows scientists to obtain and examine the regions containing the most intense events and features in the dataset in the case of turbulence. These are often the locations that have the largest vorticity norms and have intense vortices or reconnection events. In magnetohydrodynamics, the locations of largest electric current are of great interest for similar reasons. It is important that threshold queries are evaluated in an efficient manner, because often further subsequent examination and analysis is required to understand the physics that drive these intense events.

There are several challenges that arise during the evaluation of threshold queries of derived fields in an analysis database cluster. The field variables have to be evaluated *on-demand* from the array data stored in the database cluster. The evaluations are data-intensive as they perform kernel computations on extremely large multidimensional array datasets. A kernel computation computes the value at a grid location using the data points at a set of neighboring locations. Kernel computations have

to be performed at each location on the grid as opposed to at a particular number of target locations. The evaluation needs to be distributed across the nodes of the database cluster to avoid the unnecessary movement of data over the network and to achieve scalability. Techniques that target the traditional supercomputing environments do not translate directly to the distributed database setting of an analysis cluster environment.

We present a method for the efficient evaluation of threshold queries over fields derived from the raw vector or scalar fields of the numerical simulation stored in the database. Our method makes effective use of the cluster resources and achieves high throughput and scalability. We exploit the parallelism available in the cluster by means of data-parallel execution of the computations. We extend the database management system with an application-aware cache for query results. We build on the idea of an application-aware cache introduced by Lopez et al. [63] and more broadly on the concept of semantic caching [64]. Rather than caching just data as is the case in system caches and the tree-cache described by Lopez et al., we cache query results along with query metadata and subsequent queries are evaluated against the cache. This leads to query performance improvement of over an order of magnitude.

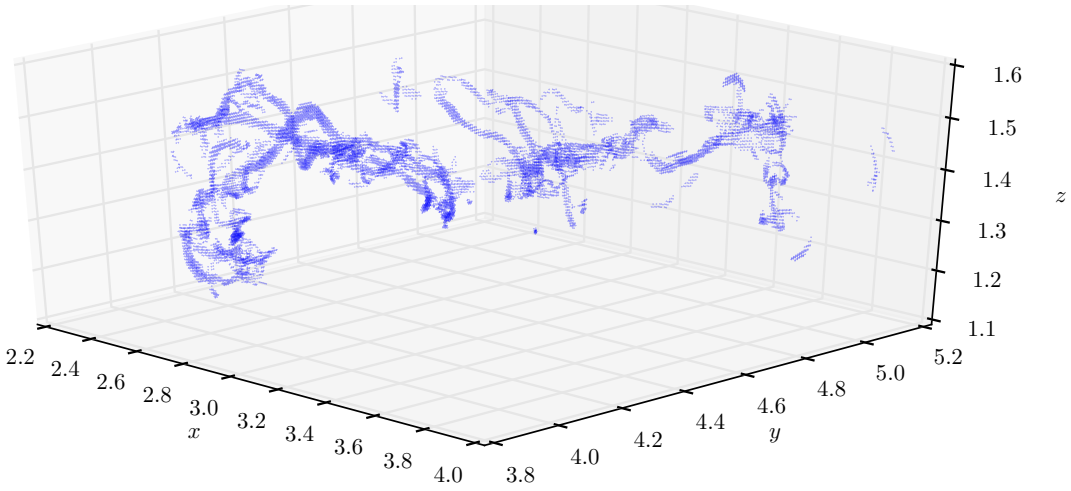


Figure 5.1: 3D (single time-step) cut through the 4D cluster containing the most intense event.

5.2 Scientific Use Cases

One of the applications of thresholding in turbulence is to find the locations of maximum vorticity in a particular time-step or the dataset as a whole. The locations of maximum vorticity are usually associated with the most intense vortices in the dataset and often have interesting and complex reconnection events associated with them. Once obtained from the service, these locations can be clustered in both 3d and 4d. This allows scientists to examine their evolution with the flow and make subsequent analysis queries as needed in order to study these events. The relationship between different "worms" (see Figure 5.1) that connect and reconnect at those locations is of the biggest interest.

The vorticity is computed from the velocity field by taking its curl:

$$\begin{aligned}\vec{\omega} = \vec{\nabla} \times \vec{v} &= \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right) \times (v_x, v_y, v_z) \\ &= \left(\frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z}, \frac{\partial v_x}{\partial z} - \frac{\partial v_z}{\partial x}, \frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y}\right). \end{aligned} \tag{5.1}$$

We use finite differencing methods of different orders for the evaluation of the curl. For example, with 4th-order centered finite differencing each partial derivative is evaluated from the 4 adjacent grid node values as follows:

$$\left.\frac{df}{dx}\right|_{x_n} = \frac{2}{3\Delta x}[f(x_{n+1}) - f(x_{n-1})] - \frac{1}{12\Delta x}[f(x_{n+2}) - f(x_{n-2})], \tag{5.2}$$

where f denotes any one of the three components of the velocity and Δx is the width of the grid in the x direction. The partial derivatives along y and z are computed in the same fashion. Figure 5.2 shows the distribution of the values of the norm of the vorticity field in the MHD dataset for a representative time-step. This is indicative of how the values are distributed in the dataset as a whole. This coarse view of the data can be used by scientists to guide the selection of threshold values.

Figure 5.1 shows the most intense event observed in the forced isotropic turbulence dataset. The locations of maximum vorticity in the dataset were clustered in this case in 4d using a friends-of-friends algorithm. It is interesting to note that the cluster containing the most intense event in the entire dataset develops from nothing (i.e. it does not appear in the first few time-steps) and it takes less than the timespan

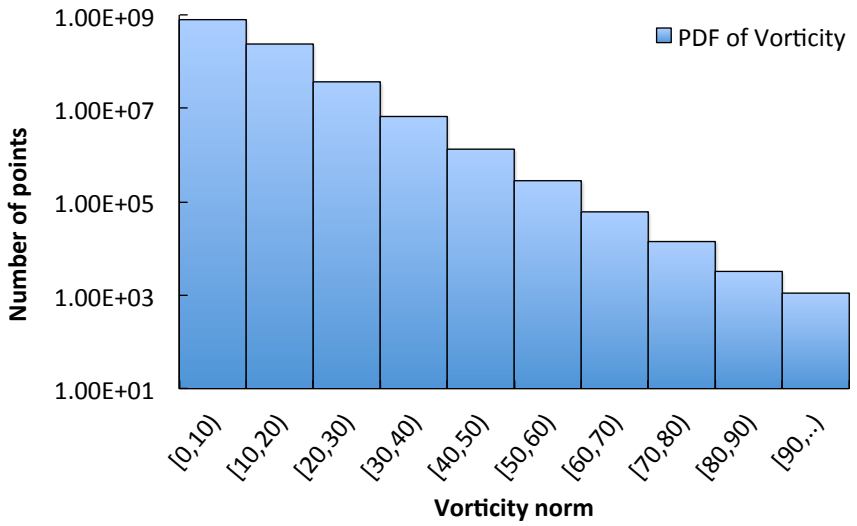


Figure 5.2: Probability density function of the norm of the vorticity field for a representative time-step for the MHD dataset.

stored in the database for it to develop. Figure 5.1 also shows that most interactions between worms are not simple. There are several worms interacting in a complex way at the same time. Similar type of analysis and the fact that the entire time history of the simulation is available in a database cluster, which provides built-in sophisticated analysis routines revealed flux-freezing breakdown in MHD turbulence [4], showing why solar flares last minutes rather than the millions of years that conventional theory would predict.

In addition to obtaining the regions of largest vorticity, there is substantial interest in studying the regions with highest values for other fields, such as the second and third velocity gradient invariants (Q and R). These invariants are scalar quantities whose values contain information about the topology of the flow and the rates of vortex stretching and rotation. In MHD, finding the locations with largest values for

the electric current can lead to new insights into the development of the most intense reconnection events of magnetic field sheets in the simulated plasma. Similarly to the vorticity, the electric current is derived from the magnetic field by taking its curl. The list of fields of interest, on which scientists would like to perform threshold queries certainly does not stop here and is indicative of how valuable this functionality is in the study of turbulence and fluid dynamics.

5.3 Threshold Query Evaluation

Threshold queries of derived fields submitted to the JHTDB are evaluated using a data-parallel execution strategy and the query results are cached in an application-aware semantic cache. In addition to query results, the cache stores their semantic descriptions and query metadata and parameters used to obtain them. The evaluation strategy for queries that do not hit the cache is driven by the spatial partitioning of the data across the nodes of the cluster.

5.3.1 Derived Fields Computation

The databases store only the raw field data from the simulation (e.g. velocity, pressure, magnetic field etc.). However, the threshold queries of most interest to science users produce all locations where the values of a *derived* field are above a given threshold. Thus, the derived field in question has to be computed from the raw

data first. For most derived fields of interest, this computation has local support. It has an associated localized kernel of computation around each grid node. Therefore, the value of the derived field at each grid node depends on the value of the stored field at all of the grid locations, which are part of the kernel of computation.

5.3.2 Distributed Data-parallel Execution

In most cases threshold queries operate over the entire stored data volume of a simulation time-step. Each such query is subdivided by the mediator into queries submitted to each of the database nodes. Each node evaluates the query over the data that it has stored locally. Only a small amount of data along the boundary need to be requested from adjacent nodes. The size of the band of data that may not be available locally is equal to a kernel half-width. Such a band is needed on each of the sides of the box forming the domain of the computation. The data are read into memory and the particular field requested is computed at each of the locations on the grid. The same strategy applies when utilizing multiple processes per node. The norm or absolute value of the derived field at each location is compared against the specified threshold and if the value is higher, it is maintained along with the spatial coordinates of the location in a list.

We impose a limit on the maximum number of locations that can be returned as a result of a threshold query in order to prevent having to return the entire data volume of a time-step or a significant fraction of it for queries with thresholds that are set

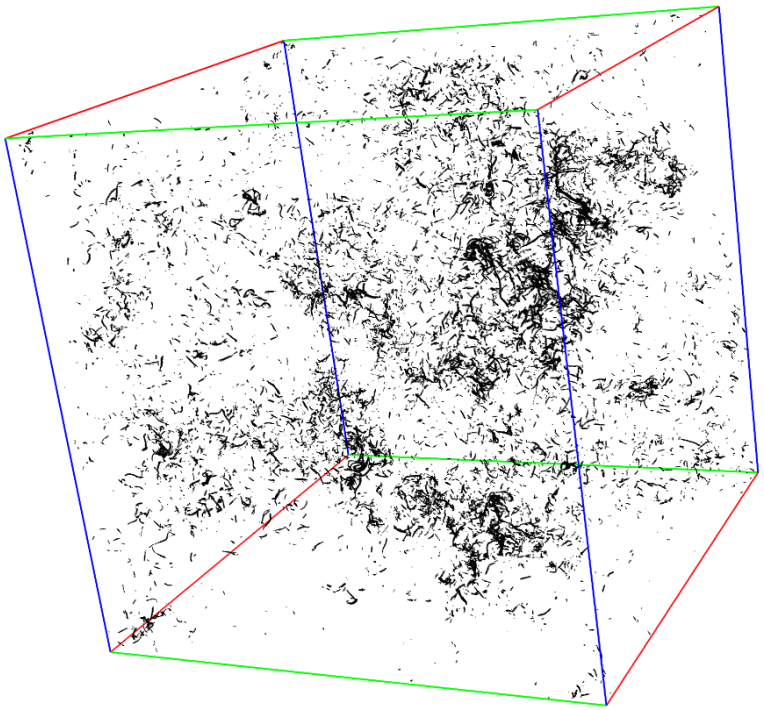


Figure 5.3: Points with values above 7 times the root mean square value of the vorticity for a single time-step.

too low. Currently this limit is set conservatively to 10^6 locations, which is sufficient to examine a time-step in detail. In the case of the vorticity, the values above 8 times the root mean square value, which is about 25% of the maximum, are contained within 2.6×10^5 points in each time-step. Figure 5.3 shows all the points in a single time-step with values above 7 times the root mean square value. There are 2.4×10^5 points in the figure. Given that we are interested in extreme events, obtaining the locations with values even within 50% of the maximum would be sufficient. At the same time this also limits the amount of data that have to be returned to the user over the network as well as the amount of data that have to be cached. Users receive an error message notifying them if their request has a threshold that is set too low.

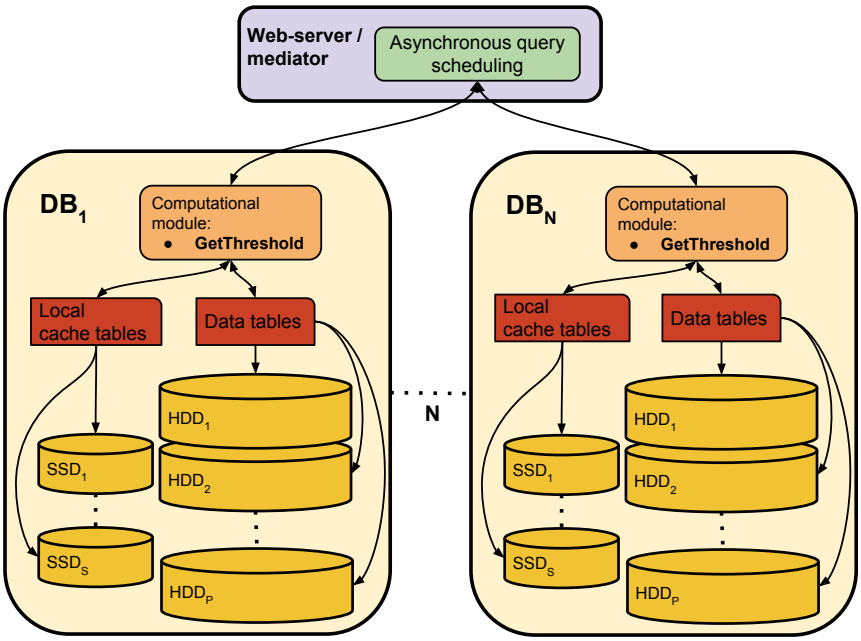


Figure 5.4: Distributed evaluation of threshold queries and architecture of the application-aware cache. Each database node has local cache tables, which reside on solid-state drives attached to the node.

If a user is interested in obtaining more data he or she can request the values of the derived field directly. Alternatively, if they are interested in the density distribution of values they can examine the probability density function (e.g. Figure 5.2), which is computed using a similar strategy to threshold queries.

5.3.3 Application-aware Cache for Query Results

A central part of the evaluation strategy for threshold queries that we have developed is the introduction of an application-aware cache for query results (Figure 5.4). The results of these queries are small compared to the amount of data that need to be examined and the results can be used to answer subsequent queries as long as they are

within the same region and specify the same or higher threshold. Each database node has a local cache. Cache entries are indexed by the field, time-step, spatial region and the threshold requested. We use a least recently used cache replacement policy. All modifications of and queries to the cache are executed within a transaction with snapshot isolation level to avoid dirty-reads or an inconsistent view of the cache.

Caching the query results preserves the computational effort in addition to substantially reducing I/O. The cached data are for the particular derived field that was queried and not the raw data of the simulation fields. Thus, we do not have to derive the requested field from the raw data for queries that hit the cache. This results in a substantial improvement in query performance as we only have to scan a small set of data and do not need to perform any additional computation.

Not all query results are suitable to caching. Most of the queries submitted to the JHTDB other than threshold queries request data at a collection of target locations. Given that there are 1024^4 possible locations for three of the datasets and $6 * 1024^4$ locations for the channel-flow dataset the chance of reuse for the results of these queries is extremely small. This is why the cache currently stores only the results of threshold queries. Nevertheless, it can easily be extended to cache the results of other query types as well if that becomes advantageous.

The cached query results are stored in a table in the database and the overall size of the cache is limited by the amount of available SSD disk space, not memory. Given a limit of $\sim 10^6$ points per time-step for a threshold query, the space required to cache

the maximum number of points including the index space and database overhead is ∼40MB. Therefore for a dataset containing 1024 time-steps, as is the case for the isotropic turbulence and MHD datasets part of the JHTDB, a cache size of 40GB is sufficient to cache the query results for threshold queries of a derived field over the entire dataset. The currently available SSD disk space per node is ∼200GB, which will be sufficient to maintain the threshold results for nearly five derived fields over the entire dataset. In contrast, computing and materializing a scalar derived field for the entire dataset would require ∼5TB (15TB for vector fields).

The entire cache is comprised of two database tables. The *cacheInfo* table stores metadata for the cached entries. It stores information about the dataset, field, time-step, start and end coordinates of the spatial region examined and the threshold value used. The *cacheData* table stores the locations of all of the grid points, for which the field queried has a norm higher than the specified threshold. The *cacheData* table is foreign key constrained with the ordinal of the *cacheInfo* table. This allows us to quickly find a record in the *cacheInfo* table and retrieve all of the cached entries using an index lookup.

5.3.4 Overall Execution of Threshold Queries

Algorithm 1 illustrates the process of obtaining all points with norms of the specified field above the given threshold from the database in the presence of a cache. The mediator submits a query to each of the database nodes storing the raw data

Algorithm 1 Get points above threshold using cache

Require: Dataset d, Field f, Timestep t, Threshold k, Query box $q = [x_l, y_l, z_l, x_u, y_u, z_u]$

```
 1: procedure GetThreshold
 2:     points ← List()
 3:     updateCache ← false
 4:     query ← SELECT * FROM cachedb..cacheInfo
                WHERE dataset = d AND field = f
                AND timestep = t
 5:     command ← SqlCommand(query)
 6:     reader ← command.ExecuteReader()
 7:     if reader.HasRows() then
 8:         k_s ← reader["threshold"]                          ▷ Stored threshold
 9:         start ← reader["startIndex"]
10:         end ← reader["endIndex"]
11:         ordinal ← reader["ordinal"]
12:         if k ≥ k_s  &  q ∈ [start, end] then
13:             query ← SELECT * FROM cachedb..cacheData
                        WHERE cacheInfoOrdinal = ordinal
14:             command ← SqlCommand(query)
15:             reader ← command.ExecuteReader()
16:             while reader.Read() do
17:                 location ← reader["zindex"]
18:                 norm ← reader["dataValue"]
19:                 if norm ≥ k  &  location ∈ q then
20:                     points.Add(new Point(location, norm))
21:                 end if
22:             end while
23:         else
24:             updateCache ← true
25:         end if
26:     else
27:         updateCache ← true
28:     end if
29:     if updateCache then
30:         Retrieve data covering q from DB.
31:         for all p ∈ q do
32:             Compute f at p.
33:             if ‖f(p)‖ ≥ k then
34:                 points.Add(new Point(p, ‖f(p)‖))
35:             end if
36:         end for
37:         Update cacheInfo and cacheData tables.
38:     end if
39:     return points
40: end procedure
```

asynchronously. Each node begins evaluation of the query by executing Algorithm 1. First a cache lookup is performed. If the data for the requested field, time-step and spatial region are available in the cache and if the specified threshold is higher than the one stored in the cache the query can be answered from there. The records are retrieved from the cache and the ones that have a higher value are returned to the mediator and subsequently back to the user. If the data stored in the cache have a higher threshold than the one requested the cache needs to be updated. Similarly, if the cache does not have an entry for the specified parameters the query needs to be evaluated from the raw data. In those cases the raw data are read into memory with data along the boundary requested from adjacent nodes as needed. The specified field is derived at each location on the grid and the norm or absolute value of the field is compared against the threshold. The locations where the values are higher than the threshold need to then be stored in the cache. If the cache does not have enough space for the new records, space is freed up by removing the least recently used data across all quantities. Reading from, updating or modifying the cache is done within a transaction with snapshot isolation level. Snapshot isolation allows us to avoid locking the tables that serve as the cache for each transaction. This provides for a higher degree of parallelism and avoids any potential deadlocks from queries running in parallel.

5.4 Experimental Results

We evaluate the developed method for the execution of threshold queries to large numerical simulation datasets with the goal of analyzing the benefits and overhead from the introduction of the application-aware cache. We also analyze the scaling properties of the method. Finally, we show that an integrated method that performs the evaluation on the database nodes near the data is several orders of magnitude faster than the user requesting the derived filed of interest from the database and evaluating the threshold locally.

5.4.1 Experimental Setup

The experiments were run on the production database nodes of the JHTDB through a development Web-server hosting the Web-services. We used the MHD dataset (Section 2.4.1) for the experimental runs. This dataset is partitioned across 4 database nodes according to spatial regions in the Morton z-order. The database nodes are 2.66 GHz dual quad-core Windows 2008 servers with SQL Server 2008 R2 and 24 GB of memory. Each node has 24 2TB SATA disks arranged as a set of four RAID-5 arrays. The database files are striped across the nodes and their associated disk arrays. The tables storing the data are partitioned spatially along contiguous ranges of the Morton z-curve and the data for each partition reside in one database file.

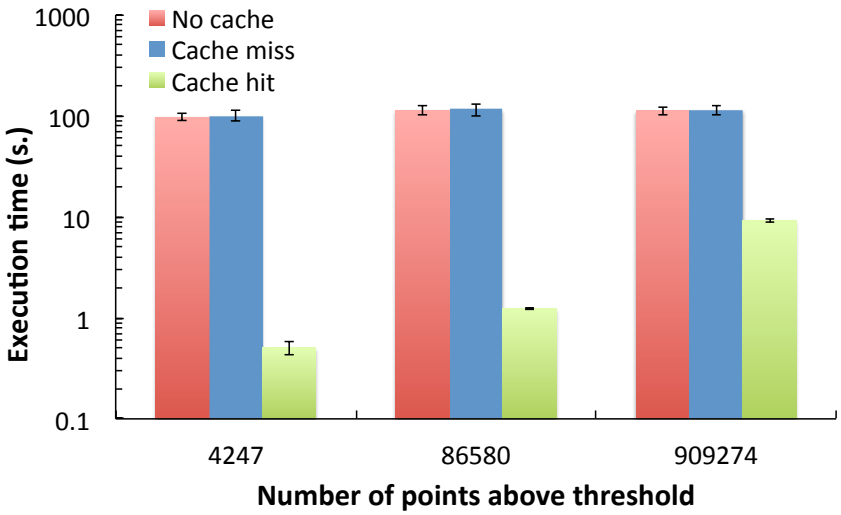


Figure 5.5: Execution time for threshold queries at different threshold levels compared with the execution time of the same queries in the absence of a cache.

For this evaluation we looked at the performance of threshold queries to the vorticity field. The vorticity field is representative of derived fields that have to be computed from the stored data. It is defined as the curl of the velocity field. As described in section 6.3 thresholding the vorticity field is important in the study of fluid dynamics and obtaining the locations of maximum vorticity can lead to new insights into the development of the most intense vortices observed in the dataset.

5.4.2 Evaluation of Cache Effectiveness

The central part of the strategy that we have developed for the evaluation of threshold queries of derived fields is the application-aware cache, which stores the results of these queries. We first evaluate the overhead associated with the introduction and maintenance of the cache. Figure 5.5 compares the execution time of queries in

the absence of a cache with the execution time of the same queries, which interrogate the cache first (blue and red bars in the figure). The execution times are also shown in Table 5.1. For these experiments we requested the locations with norms of the vorticity above thresholds at different levels. We refer the reader to Figure 5.2, which shows the distribution of values of the norm of the vorticity field in the MHD dataset to get an appreciation of the different threshold values used in the experiments. For the first set the threshold was set high (80.0) and only ∼4,300 points (or 0.0004% of all points) were above the threshold. For the second set a medium threshold (60.0) was chosen and ∼87,000 points (or 0.0081% of all points) were above the threshold. Finally, a low threshold (44.0) was chosen for the last set and there were ∼900,000 points (or 0.0847% of all points) above the threshold. For each set a random time-step was chosen and the queries were run against that time-step. The measurements were taken from the point of view of the end user.

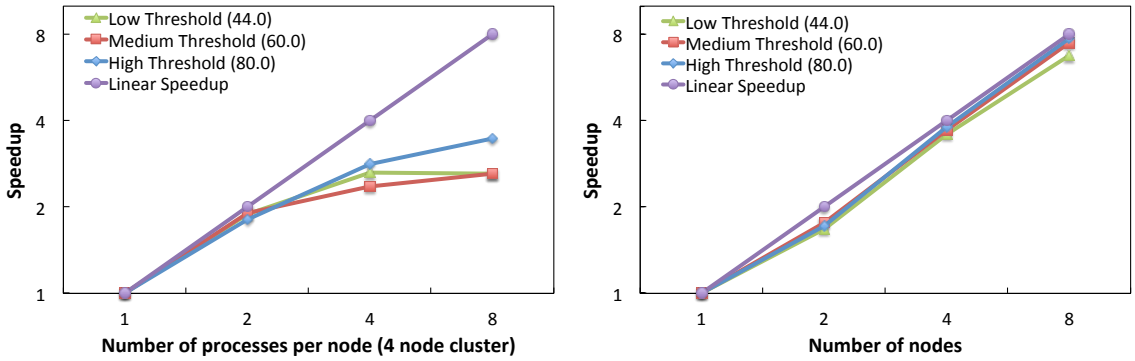
As we can see from the results shown in Figure 5.5 the overhead associated with querying the cache first is minimal, less than 3% and within the margin of error. The cache was initially populated by executing several hundred unrelated queries and contained several million entries. During the "cache-miss" runs cache entries for the particular time-step queried were dropped before each run, making sure that each query would produce a cache miss and would have to be evaluated from the raw data. The execution times were averaged over 10 runs. We utilized 4 processes per database node for the evaluation of each query. The method shows stable running time across

different time-steps and threshold levels in the absence of a cache and during cache misses. The running time increases slightly only because of the larger result set that has to be returned to the user.

Vorticity threshold	Points above threshold	Average Running time (s.)		
		No cache	With cache (miss)	With cache (hit)
80.0	4247	97.1	100.2	0.5
60.0	86580	113.7	115.9	1.2
44.0	909274	111.6	115.0	9.1

Table 5.1: Effectiveness of caching.

Cache hits reduce the running time of threshold queries by over an order of magnitude as shown in Figure 5.5 and Table 5.1. This is because we do not have to compute the requested derived field from the raw data, which eliminates the associated I/O. Only the cache entries need to be looked-up, which is substantially less data than the raw vector or scalar field data. For the queries with large result sets it is actually the network time taken to transfer the results to the user that dominates the overall execution as opposed to the I/O or computation time as we show later. Cache hits are evaluated by first warming up the cache by submitting the same set of threshold queries of the vorticity field as before. We then submit several more unrelated queries with different time-steps and threshold values in order to pollute the cache. Finally, we issue the original set of queries and measure their running times. Let us focus on the query with low threshold, which returns ∼900,000 points. Given that valid threshold values are limited to those that result in no more than 1,000,000 points it



(a) Scale-up with multiple processes per node (b) Scale-out to multiple nodes

Figure 5.6: Execution time for threshold queries at different threshold levels – high, medium and low. The scale-up evaluation was performed utilizing 1-8 processes per server on a 4-node cluster. The scale-out evaluation was performed on 1 through 8 nodes.

is likely that all subsequent queries to this time-step will result in a cache hit as their threshold is likely to be equal or higher than the cached one. Currently we observe fairly high cache-hit ratios as the workload is very structured and queries tend to examine the same regions in space and time.

5.4.3 Scaling and Distributed Evaluation

The evaluation of threshold queries of derived fields from the raw data is both I/O and computationally bound. These queries examine the entire data volume of a simulation time-step and are, therefore, good candidates for a data-parallel distributed evaluation. Our data-parallel implementation exhibits good vertical and nearly ideal horizontal scaling as shown in Figure 5.6. For the scale-up experiments (Figure 5.6(a)), we used the same queries and threshold values as for the runs shown

in Figure 5.5 and Table 5.1 but with varying number of processes per node. Cache entries for the time-step queried were again dropped before each run in order to evaluate the scaling properties of the computation of the derived field from the stored data. The computations for all of the derived fields of interest (such as the vorticity) at each grid point need data from adjacent grid points only. Therefore, each node of the cluster is able to compute the derived field from data available locally with only a small amount along the boundary of each region having to be retrieved from adjacent nodes. Each computation is independent and embarrassingly parallel. This allows us to make use of multiple processes per node and scale out to multiple database nodes.

We observe nearly a two times speedup when going from a single process per node to two processes per node (Figure 5.6(a)). The speedup diminishes to 1.4 times when going to 4 processes and little speedup is observed with 8 processes per node. While the computation time scales with increased process count, the time to perform I/O does not as the data on each node reside in the same database table and on the same set of disks. Additionally, I/O redundancy increases as the process count increases as data along the boundary of each region are requested by multiple processes. SQL Server already utilizes parallelism to perform the I/O even when data are retrieved utilizing a single query. Finally, the experiments were run on the live production database nodes, which were also servicing other user queries in addition to operating system and other SQL Server processes. Nevertheless, running with 4 processes per node is nearly 2.6 times faster when compared to running with a single process.

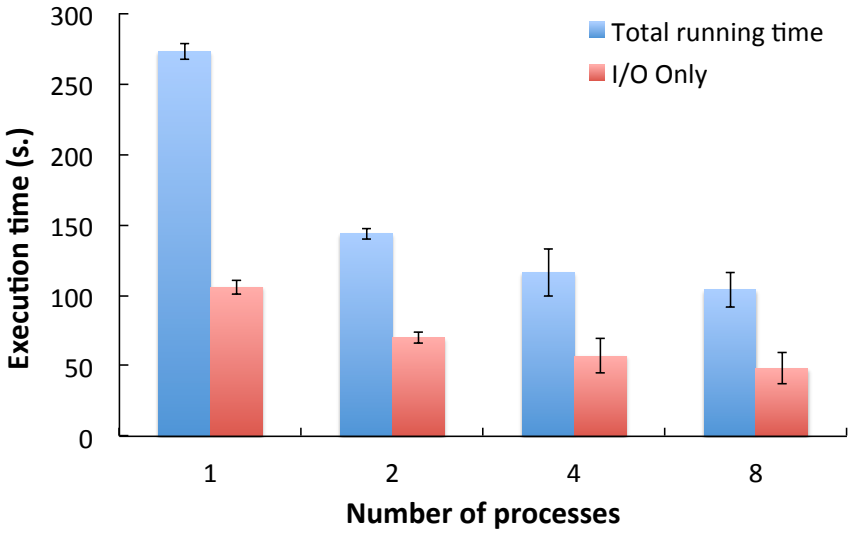


Figure 5.7: Execution time for threshold queries evaluated utilizing different number of processes per server compared with the time taken to perform the I/O only.

The scale-out experiments show a nearly perfect linear speedup as the evaluation is distributed to an increasing number of database nodes (Figure 5.6(b)). For these experiments we issued queries with the same threshold levels as before to a cold cache. We utilized a single process per database node to evaluate the horizontal scaling of the computation. The evaluation benefits not only from the additional computational resources with the addition of database nodes to the cluster but also from the increased memory size. The data needed for the computation of each derived field are read into memory and the larger memory size means that there is less contention with other system and application processes and it is less likely that virtual memory needs to be used. SQL Server also benefits from a larger buffer pool, which reduces the I/O time.

As expected, we observe even weaker speedup when the queries perform nothing but I/O and the number of processes per node is increased. Figure 5.7 compares

the running time of the queries with a medium threshold and executed with varying number of processes per node with the time taken to perform the I/O only. The I/O time is about half of the total running time for these queries. SQL Server already makes use of parallelism internally and the data have to be retrieved from the same set of disks. Nevertheless, the I/O time does decrease with additional processes, this is because the data reside in a partitioned table and the data in each partition are placed in a separate file on one of the disk arrays. Depending on how the data requests are scheduled in SQL Server this allows for the disks arrays to be driven in parallel. Additionally, with more processes per node the data can also be consumed faster. It is worth noting that the total running time for the queries evaluated with 4 or 8 processes is about the same as the time it takes to perform the I/O only with a single process.

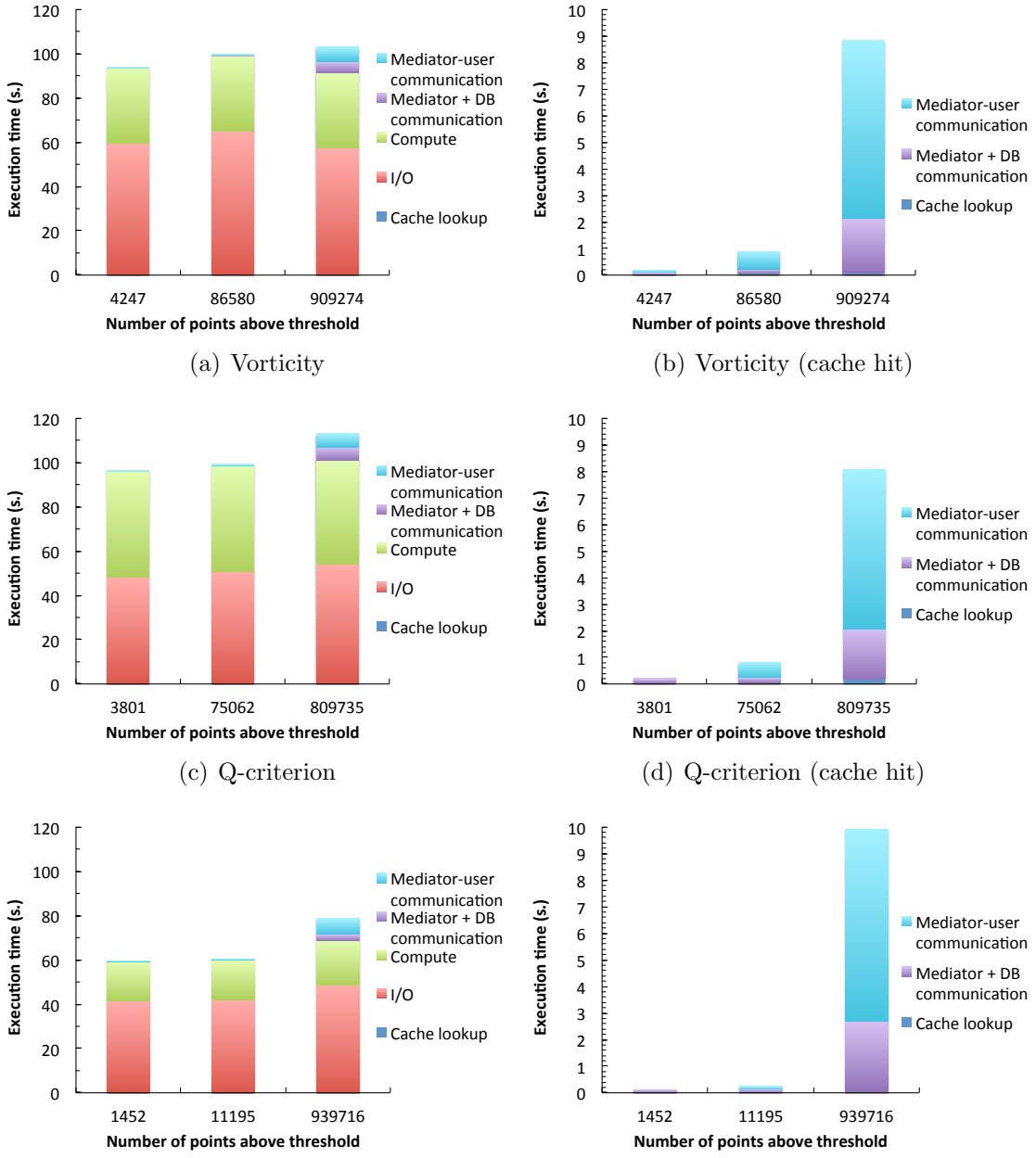
So far we have presented the effectiveness of evaluating threshold queries of derived fields on the database cluster storing the raw simulation data. The data-parallel computation of the derived fields allows us to evaluate a threshold query over an entire 1024^3 time-step part of a 20TB dataset in less than two minutes. The introduction of an application-aware cache for the query results of these queries reduces this time to several seconds when there is a cache hit. In contrast, one of our science collaborators reported that his evaluation of this functionality performed locally would take over 20 hours to complete. To perform the evaluation locally the user requests the derived field of interest from the database by submitting multiple queries over subregions of a

time-step. This is necessary as requesting a derived field over the entire data volume of a particular time-step will overload the network. Derived fields may have even more components than the scalar or vector field stored in the database. For example, the velocity gradient (needed for the computation of the vorticity) has 9 components compared with the 3 components of the velocity. Given a single-precision floating-point representation, this makes the velocity gradient of an entire time-step at least 36GB in size. A Web-service request will be much larger due to the overhead of wrapping the data in an xml format. After the field of interest is obtained locally the user has to threshold it to get the final result, which is reasonably fast, but discards most of the data that have been requested to yield a small in size result.

5.4.4 Evaluation of Additional Fields

The data-parallel evaluation of threshold queries shows stable execution time for different derived fields in addition to the different threshold levels and time-steps queried. The execution times depend on the complexity of the computation needed to evaluate the particular derived field requested. Figures 5.8(a), 5.8(c) and 5.8(e) show a breakdown of the execution time of threshold queries of different derived fields, which are evaluated from the raw data and a cold cache using 4 processes per node on a 4 node cluster. Almost the entire time is spent performing the I/O and computation associated with the derived field requested.

The vorticity field and the Q-criterion have similar I/O requirements as they have



(a) Vorticity

(b) Vorticity (cache hit)

(c) Q-criterion

(d) Q-criterion (cache hit)

(e) Magnetic field

(f) Magnetic field (cache hit)

Figure 5.8: Breakdown of the execution time for threshold queries requesting different fields and at different threshold levels – high, medium and low.

the same kernels of computation and are both derived from the velocity gradient. The vorticity has 3 components and its computation only examines 6 of the 9 components of the velocity gradient, which are also examined in pairs (see Equation 5.1). On the other hand, even though the Q-criterion is a scalar value, it is computed through a non-linear combination of all 9 of the components of the velocity gradient. This means that the velocity gradient has to be computed at each grid location before the Q-criterion is evaluated, which is reflected in the increased computation time that we observe for the Q-criterion. The magnetic field is one of the raw fields of the magnetohydrodynamics dataset that are stored in the database. Therefore, there is no additional computation needed to derive it from the data, every data point has to be simply compared with the threshold level specified. This is why the computation time is much smaller compared to the queries for the vorticity and the Q-criterion. The I/O time for the magnetic field is also smaller. This is because its kernel of computation is a single point and therefore there are no additional data along the boundary that have to be requested from adjacent nodes. In that case all of the data needed are available locally for each database node.

In all of these cases, the time taken to interrogate the cache is negligible. The mediator time to distribute the queries and assemble the results as well as the time to transfer them to the user are also substantially smaller than the I/O and computation times. As expected they increase proportionally to the number of points in the result set.

It is interesting to note that the time taken to perform a cache lookup is relatively small even in the case of a cache hit as can be seen in Figures 5.8(b), 5.8(d) and 5.8(f). This is because the cache tables reside on SSDs attached to each database node (see Figure 5.4) and retrieving the data is always done through a clustered index lookup. In the cases of a cache hit, the majority of the time is spent simply transferring the results from the database nodes to the mediator and then back to the user. These times remain more or less constant between the cases of cache misses and cache hits (left and right column of Figure 5.8). Caching the results of threshold queries effectively preserves the I/O and computational effort spent during their initial evaluation and results in over an order of magnitude speedup for all the different fields requested as we can see in Figure 5.8.

5.5 Related Work

Only select few database systems offer support for arrays as first-class citizens. Even fewer provide the fault-tolerance, scalability and availability guarantees necessary for a system managing multi-terabyte datasets in a production setting. This is part of the reason why we have chosen to represent the array data in the JHTDB as a collection of binary large objects in a relational DBMS and perform the array manipulation tasks necessary at the application level. The systems that provide support for arrays and aim to handle large array data efficiently are RasDaMan [65],

SciDB [66] and MonetDB/SciQL [67]. RasDaMan partitions raster objects into tiles, which are stored in a traditional relational database system. This approach is similar to how the numerical simulation data are handled in the JHTDB. RasDaMan provides RasQL [68], which is a SQL-92 based query language for the manipulation of raster images. SciDB is an array database system build from the ground up. Array attributes are partitioned vertically and each attribute array is decomposed into overlapping chunks. SciDB provides a declarative Array Query Language (AQL) and an Array Functional Language (AFL). Users can create arrays with named dimensions with AQL and make use of the functional operators defined in AFL, such as SLICE, SUBSAMPLE, SJOIN, FILTER and APPLY. SciQL's focus is on language design and integration with SQL:2003 syntax and semantics. It is implemented within the MonetDB framework [69].

Database systems support rollup queries, including top-k queries, but in most cases these queries apply only simple linear score functions on the attribute values of individual records. Additionally, many top-k query evaluation techniques rely on the score functions being monotone in order to perform early pruning (see [70] for a survey of top-k evaluation strategies). This is an assumption that we cannot make for the functions used to compute all the different possible derived fields of interest in fluid mechanics. Even approaches that aim to work with general score functions [71, 72] assume that the function operates on the attributes of a single record. In contrast, our approach performs a kernel computation at each grid location in order

to obtain the value of a *derived* field at that location and examines the vector or scalar array data at all neighboring locations, which are within the kernel of the computation. The functions used to derive the field may even be non-linear. Finally, a top-k approach may not be suitable in the cases where scientists are interested in performing threshold queries at different time-steps as the same threshold level will produce different number of points in the result set for different time-steps.

The processing of top-k queries has been studied extensively in the context of distributed and relational database systems. A survey of different techniques in the case of centralized processing is given in [70]. In the case of distributed processing different approaches focus on horizontally [73, 74] or vertically [75, 76, 77, 78, 79] distributed data. None of these approaches deal with array data stored in a relational database system. Zhao et al. propose an algorithm for the processing of top-k queries in large-scale distributed environments called BRANCA [80]. They build on the idea of semantic caching [81] and make use of branch caches, which store results of previous top-k queries with respect to the data stored on each server. The caching mechanism that we use is similar in that regard, but the queries that are evaluated in our system operate on derived fields, which are computed at each location by accessing data from a surrounding region. The queries described in [80] operate over the attributes of individual records only using simple linear score functions.

Aßfalg et al. introduce the concept of threshold queries in time-series databases [82]. Their definition of threshold queries differs from the threshold queries described

herein. They are concerned with determining the time-series, which exceed a user-defined threshold at time frames similar to the time-series specified in the query. Thus, their definition of threshold queries is concerned with the temporal relationship between the time-series stored in the database (usually one dimensional sequence of measurements) and the time-series given in the query. In contrast, our approach focuses on reporting all of the spatial locations of a multidimensional field where the norm or absolute value of the field exceeds a user prescribed threshold.

In a system called the tree cache, Lopez et al. [63] make use of a small application-aware cache to reduce access time to large datasets stored on disk. The tree cache stores individual octants of octree datasets and exploits application-specific information to determine which octants to cache and to perform query reordering. This work has inspired the use of an application-aware cache for the evaluation of threshold queries. In contrast to the tree cache, we do not cache raw data objects, but rather query results. Caching query results preserves the computational effort in addition to reducing I/O, which has a much bigger impact on query performance and substantially reduces the size of the cache. Additionally, the cache that we introduce resides on disk rather than in memory, which greatly increases its potential size. Lopez et al. also explore approximate querying through aggregation, which can be fairly easily supported by our system but is of limited use as scientists performing threshold queries are usually interested in obtaining the exact locations where a field is at its highest values.

Sampling approaches [83, 84] offer an alternative to the on-demand computation of derived fields and the evaluation of threshold queries on them. The goal of both techniques is to not return large data volumes, but focus on the most intense events and interesting regions in the dataset. The computation of derived fields is carried out on the nodes of the database cluster and takes a look at the dataset as a whole, while the user obtains only a small subset of the data, where the derived field in question is above the prescribed threshold. Sampling approaches can potentially omit some locations and while useful for generating initial impressions may not be suitable if the exact locations where a field is at its highest values are desired.

Andrade et al. [85] describe a database system and an optimization framework build on the concept of *active semantic caching*. An active semantic cache aims to fully or partially reuse cached query results or aggregates through automated transformations of these aggregates. Similarly to our work they focus on real scientific data-analysis applications. The method that we have developed for the evaluation of threshold queries complements the active semantic caching approach and could be used in that framework. Our work has focused on extending a relational database system (Microsoft's SQL Server) as opposed to designing a new database system from the ground up as described by Andrade et al. [85].

5.6 Discussion

We have presented an efficient strategy for the evaluation of threshold queries of derived fields in large numerical simulation datasets. The thresholded fields are derived from the stored simulation data in a distributed data-parallel manner. The computations scale with the cluster resources and are performed on the database nodes, where the data are stored. This new capability allows researches to quickly obtain and focus on regions of special interest even if they lack the computing capabilities or data transfer rates necessary to examine entire time-steps or large parts of the entire dataset.

We have introduced an application-aware cache for the query results of threshold queries. Cache hits reduce query running times by over an order of magnitude. The cache adds minimal overhead during the evaluation of queries even if there is a cache miss and has modest storage requirements. The cache is represented as a set of database tables and resides on disk rather than in memory. Each database node has local cache tables, which allows the cache to scale-out as the cluster grows.

Chapter 6

Particle Tracking in Open Simulation Laboratories

Particle tracking along streamlines and pathlines is a common scientific analysis technique, which has demanding data, computation and communication requirements. It has been studied in the context of high-performance computing due to the difficulty in its efficient parallelization and its high demands on communication and computational load. In this chapter, I present a study of efficient evaluation methods for particle tracking in open simulation laboratories [86]. Simulation laboratories have a fundamentally different architecture from today's supercomputers and provide publicly-available analysis functionality. In this study, my collaborators and I focus on the I/O demands of particle tracking for numerical simulation datasets 100s of TBs in size. We compare data-parallel and task-parallel approaches for the advection

of particles and show scalability results on data-intensive workloads from a live production environment. We have developed particle tracking capabilities for the forced isotropic turbulence, magnetohydrodynamics and homogeneous buoyancy-driven turbulence datasets of the JHTDB.

6.1 Motivation

Particle tracking is most often thought of as a technique for visualizing flow fields. It is one of the most powerful techniques for this task. However, it is also an *analysis* technique and an important tool used in the study of fluid dynamics. It has been used to study Lagrangian velocity structure functions and tensor-based Lagrangian time correlation functions [87]. It has been used to generalize previous models of turbulent dispersion [88]. It has been used to explore the breaking of time-reversal-symmetry in turbulence [89]. Many other examples are readily available. The efficient evaluation of particle advection in scientific analysis cluster environments therefore deserves as much attention as particle tracing in supercomputing environments for the purposes of visualization. We study efficient evaluation techniques for the advection of particles in an open numerical simulation laboratory. The analysis cluster environment does not have the parallelism available in a supercomputing facility, but instead achieves high aggregate throughput based on I/O and network bandwidth. The numerical simulation datasets stored in the analysis database cluster are much

larger than the amount of distributed memory available and therefore I/O becomes the main consideration as opposed to communication and computation.

Open simulation laboratories, such as the JHTDB, aim to provide public access to world-class numerical simulation datasets as well as built-in analysis functionality, implemented following the "move the computation to the data" paradigm [59]. It has already been shown that performing the analysis on the database nodes where the data are stored is much more efficient than transferring the data over the network [10] and particle tracking is no exception. However, in the case of particle tracking the data requirements are not static and depend on the positions computed from each preceding iteration. This means that the data necessary to compute particle positions may reside on different database nodes during the advection process. Therefore, when particles cross the data boundary of a database node they either need to be reassigned or the data need to be retrieved from adjacent nodes. A straightforward approach that synchronizes the execution at a mediator level avoids this issue by redistributing particles to database nodes during every step of the process. However, this results in the sometimes unnecessary movement and reassignment of particles between the mediator and the database nodes. Such an approach also does not allow for asynchronous processing as particles are advected together in a batch and there is effectively a barrier after each iteration step.

Particle tracking functionality was first implemented in the JHTDB in 2011 [87] and there have been nearly 6 million particle tracking requests made to the service

since then. A total of 12.7 billion particles have been advected through the stored simulated flows with an average request size of ∼2200 particles. This functionality has facilitated a number of research tasks, which have resulted in numerous publications, including research on turbulent dispersion [4, 88, 90], shape evolution [89], particles-turbulence interactions [91], calibration of experimental techniques [92] and others. The initial implementation adopted the simple straightforward approach of synchronizing particles after every iteration step at the mediator level, redistributing them to the database nodes, where the data are retrieved and next particle positions computed. The advantage of this approach is that particles are evaluated on each database node as a batch and the data for all particles are retrieved using a single database query. No data need to be retrieved from adjacent nodes and particle positions are evaluated using the efficient I/O streaming approach, described in Section 3. Nevertheless, this approach has drawbacks as well, the majority of particles may not need to be reassigned to a different database node after each iteration step, which results in unnecessary communication between the mediator and the database nodes.

Previously particle tracking was studied mostly in the context of visualization of scalar or vector fields in scientific data and on high-performance computing (HPC) architectures. Existing approaches rely on the massive parallelism available in supercomputers and in many cases avoid the I/O bottleneck by reading the entire dataset into memory [93, 94]. The amount of parallelism in analysis cluster environments is much smaller and the datasets are much larger than the distributed memory available.

On the other hand, scientists are interested in immersing particles and examining their evolution with the flow at different locations and time-steps for analysis purposes and often require very accurate results. Therefore, preprocessing the entire datasets and approximation techniques used for visualization [95] are often not an option.

We study particle tracking for the purposes of scientific analysis in open simulation laboratories. We examine different approaches to the problem with a focus on the I/O requirements of the task. Our implementation provides particle tracking capabilities to scientists around the world that may not have access to supercomputers or the necessary sophistication to develop and run codes. The advection of particles is done on the database nodes where the data are stored. We show that a task-parallel implementation, which evaluates particles in batches using an I/O streaming method for the retrieval of data and evaluation of particle positions is up to 3 times faster than data-parallel asynchronous approaches or approaches that synchronize particle advection at a mediator level. We evaluate these approaches on microbenchmarks and user workload from the usage logs of the JHTDB.

6.2 Overview of Particle Tracking

Particle tracking is implemented in the JHTDB by means of a *GetPosition* function [87]. The function tracks a list of particles simultaneously and returns the final particle

locations at the end of the trajectory integration time specified by the user. Particle tracking is performed using a second order Runge-Kutta integration scheme.

The *GetPosition* function takes as parameters a list of particle locations $\mathbf{y}$, a start time (t_{ST}), an end time (t_{ET}) and a particle integration time-step (Δt_p) and returns the particle locations at the specified end time. Fluid particles can be tracked both forward in time by specifying $t_{ET} > t_{ST}$ and backward in time by specifying $t_{ET} < t_{ST}$. Particle tracking is accomplished by integrating between times t_{ST} and t_{ET} the equation

$$\frac{\partial \mathbf{x}^+(\mathbf{y},t)}{\partial t} = \mathbf{u}^+(\mathbf{y},t), \tag{6.1}$$

where $\mathbf{x}^+(\mathbf{y},t)$ and $\mathbf{u}^+(\mathbf{y},t)$ denotes the position and velocity at time t of the fluid particle originating from position $\mathbf{y}$ at initial time t_{ST} (superscript + represents Lagrangian quantities following the fluid particle). We replace the Lagrangian velocity $\mathbf{u}^+(\mathbf{y},t)$ with the Eulerian velocity from the database $\mathbf{u}(\mathbf{x},t)$ at the particle location $\mathbf{u}^+(\mathbf{y},t) = \mathbf{u}(\mathbf{x}^+(\mathbf{y},t),t)$.

To advance the particle positions between two successive time instants t_m and $t_{m+1}(= t_m + \Delta t_p)$ the second order Runge-Kutta integration scheme consists of two steps. The predictor step yields an estimate:

$$\mathbf{x}^* = \mathbf{x}^+(\mathbf{y},t_m) + \Delta t_p\ \mathbf{u}^+(\mathbf{y},t_m). \tag{6.2}$$

The corrector step then computes the particle position at t_{m+1} using the predictor

estimate and the original particle location:

$$\mathbf{x}^+(\mathbf{y}, t_{m+1}) = \mathbf{x}^+(\mathbf{y}, t_m) + \Delta t_p \, [\mathbf{u}^+(\mathbf{y}, t_m) + \mathbf{u}^+(\mathbf{x}^*, t_{n+1})]/2. \tag{6.3}$$

In order for the integration to end exactly at the specified t_{ET} the integration time-step is equal to the smaller of the specified Δt_p and the difference between t_{ET} and t_m. Particle advection proceeds until t_m reaches the user-specified final time t_{ET}. The *GetPosition* function then returns the final particle positions $\mathbf{x}^+(\mathbf{y}, t_{ET})$ for all particles that were at initial locations $\mathbf{y}$.

Accurate spatial and temporal interpolations are important to obtain the fluid velocities while tracking fluid particles. Spatial interpolation using Lagrange polynomials with various optional orders of accuracy can be specified by the user and temporal interpolation is done by default using PCHIP (see Section 2.2). This means that every step of the Runge-Kutta integration (both the predictor and corrector steps) requires a query to 4 time-steps stored in the database and every particle's position is computed using 64, 216 or 512 data points depending on the order of the Lagrange polynomials used (4th, 6th or 8th order).

6.3 Scientific Use Cases

The ability to track particles both forward and backward in time, without having to retrieve any data to a local machine or handling the distribution of particles and

their advection has allowed scientists from around the world to perform sophisticated analysis tasks using their desktops or laptops. Providing this capability for landmark turbulence simulation datasets and making it publicly available has enabled new insights in the study of turbulence. As an example, analysis of this type has revealed why the breaking and reconnection of magnetic field-lines in solar flares lasts several minutes and not millions of years as predicted by classical theory [4].

Fluid particle tracking is one of the most important techniques in the study of the Lagrangian description of turbulence, in which a large number of particle trajectories are followed in order to capture the overall space and time-scales of the flow. Studies of turbulent transport and mixing processes often make use of the Lagrangian description of turbulence. Other examples include studying entrainment processes at turbulent/laminar interfaces [96] and modeling processes such as atmospheric pollution transport and turbulent combustion [41]. Extracting Lagrangian data from stored numerical simulation data requires an iterative process, which queries multiple time-steps of the simulation. This process is complicated further by the fact that particles' spatial locations are continuously changing. In a distributed environment their subsequent evaluation may require data stored on different database nodes. The *GetPosition* functionality of the JHTDB is a valuable tool, which facilitates Lagrangian analysis and here we study efficient approaches for its implementation.

6.4 Particle Tracking Methods

We describe three approaches to the implementation of particle tracking in open simulation laboratories. These approaches share some common aspects with techniques used for the visualization of particle traces in high-performance computing environments or supercomputers. We implement these approaches in a markedly different environment, a scientific analysis cluster consisting of a sharded relational database cluster, which stores the numerical simulation data and a Web-server front-end, which hosts publicly-available Web-Services and acts as a mediator. The particle tracking functionality is moreover used not for visualization, but for analysis tasks and requires accurate spatial and temporal interpolation for the computation of each particle position.

The three approaches that we examine are a task-parallel method, a data-parallel method and a method which synchronizes execution at the Web-server/mediator level. The task-parallel approach distributes particles based on the spatial partitioning of the data, but allows them to be integrated forward on a database node even if they cross the node's data boundary. The data-parallel approach implements services running on each database node and integrates particles forward only if they are within the node's data boundary. When a particle crosses the data boundary it is reassigned to the appropriate node or nodes for evaluation. This method supports asynchronous processing of individual particles and queries data from multiple time-steps as needed. Finally, the last method which we examine is a straightforward approach to particle

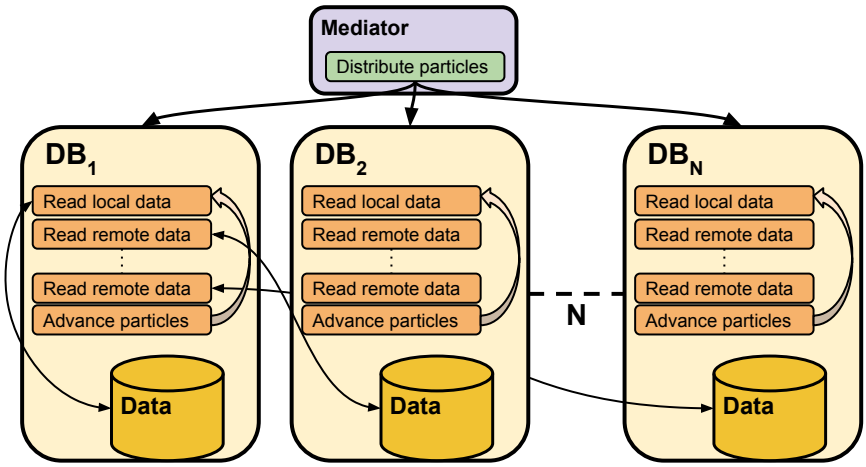


Figure 6.1: Task-parallel particle tracking approach.

tracking, which synchronizes the evaluation at the mediator level. This is also the method that was initially implemented in the JHTDB.

6.4.1 Task-parallel Evaluation

The task-parallel approach to particle tracking effectively deals with the data dependency that arises during particle advection. Namely, the fact that each subsequent integration step during the advection process depends on the results of the previous step. This approach distributes the initial particle positions to the database nodes based on the spatial partitioning of the data and each database process performs the particle advection for the entire batch of particles assigned to it until completion (Figure 6.1). No reassignment of particles is necessary and the fact that particles are advected together as a batch means that the data retrieved from the database is for the same set of time-steps during each integration step. This allows us to make

full use of the I/O streaming method for the evaluation of batch queries performing spatial and temporal interpolation (Section 3). Using this method particle positions are evaluated by means of partial-sums, which allows us to perform a single sequential read over the data.

The task-parallel approach must handle particles that have crossed the data boundary of a database node. Each process maintains a connection to all database nodes, which have data necessary for the evaluation of any given particle's position. A connection is opened only if there are particles that need data from the particular database and the data for all particles that have partially or fully crossed into a different database are retrieved as a batch. The I/O streaming method supports distributed evaluation, which allows us to evaluate particle positions even if the data are coming from different databases. Given that the data are distributed across a relatively small number of nodes (4 to 8) and the spatial volume of a dataset is large (1024^3 data points) the number of particles that cross into a different database is not large. Moreover, the scientific cluster environment is built with the goal of providing network speeds that are on-par if not faster than the aggregate disk read speeds, which means that there is little downside to the retrieval of data from adjacent database nodes.

The use of the I/O streaming method for the evaluation of particle positions during each integration step and the evaluation of all particles as a batch minimizes the I/O incurred during each step. The use of static initial distribution of particles

with the task-parallel approach aims to balance the execution around I/O. We have found that this has the biggest impact on overall execution time of particle tracking queries. Nevertheless, it is straightforward to modify this approach with the goal of balancing the computational load on each database node, by distributing equal-sized particle batches to each database node.

6.4.2 Data-parallel Evaluation

The data-parallel particle tracking approach distributes particles to processes based on the spatial partitioning of the data. Each process advects particles, for which the necessary data are available locally and reassigns particles to other nodes if they cross the data boundary. This approach allows for the asynchronous movement and reassignment of particles from process to process. Each process has to retrieve data from multiple different time-steps depending on the progress of the particles assigned to it at any given step. This approach is based on the established parallelization paradigm of static allocation of blocks to processes [97]. The difference however is that instead of reading all blocks assigned to a process into memory at the start of the computation, the data necessary for the evaluation of all particles assigned to a process are continuously read from the local database.

The data-parallel approach performs particle tracking as follows. The mediator distributes the particles to processes running on each database node based on the static spatial-partitioning of the data. The processes are implemented as services in

the Windows Communication Framework (WCF). Each process then begins advecting particles by retrieving all of the data needed for the computation of the positions of the entire batch of particles. Each process keeps track of whether a particle has crossed the local data boundary or whether a particle has reached the end time and its advection is therefore complete. Particles that have crossed the local data boundary are reassigned to the appropriate process by means of a one-way send. Particles that have partially crossed a data boundary are assigned to multiple processes for evaluation and the partial results are sent back to the process that owns the particle. Finally, when a particle has reached the specified end time it is sent back to the mediator. When the mediator receives all particles it sends the final positions back to the user.

One of the drawbacks of the data-parallel approach is that because of the asynchronous nature of the movement of particles between processes, each process has to retrieve data from multiple different time-steps at any given time. This increases the overall I/O and subsequently execution time. Additionally, for workloads that result in a large number of particle crossings, the movement and reassignment of particles increases the communication costs. Recently, domain traversal approaches that make use of data-parallelism have shifted towards group-based communication flows in order to alleviate some of these issues. For example, the *DStep* system [93] makes use of worker groups with an assigned communicator, which batches communication requests and executes them in a group-based manner.

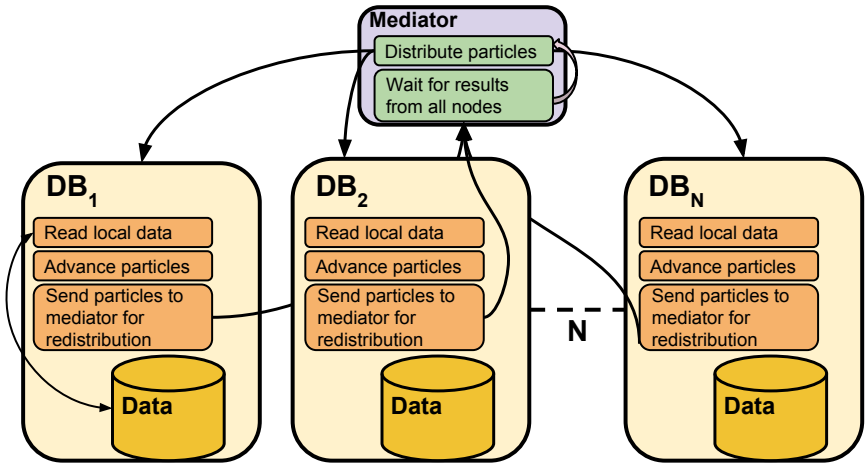


Figure 6.2: Synchronization of particle integration at the mediator.

6.4.3 Synchronization at Mediator Level

Similarly to the data-parallel particle tracking approach, the last method that we examine distributes and advects particles only to processes that have the data necessary for each integration step available locally. However, this approach achieves this goal by different means. In order to support the batch execution of particles it sends particle positions back to the mediator after each integration step and the particles are then redistributed based on their new positions. This mode of execution effectively places a barrier after each integration step, which guarantees that each process has all of the data necessary for particle advection available locally and can retrieve them by means of a single sequential read. Each process computes the particle velocities and the new particle positions using the I/O streaming method for the execution of batch queries (Section 3). The process is shown in Figure 6.2.

This is a straightforward approach to particle tracking and this is the method that

was initially implemented in the JHTDB for the *GetPosition* function. This method guarantees optimal I/O for each integration step. However, this comes at the cost of increased communication between the mediator and the database processes performing the integration. All particles are sent back to the mediator after each integration step even though the majority of them may not have crossed the data boundaries of the process they are assigned to and therefore do not need to be reassigned. Additionally, the mediator synchronization means that particle advection proceeds at the pace of the slowest process.

6.5 Experimental Results

We evaluate the performance of the different particle tracking approaches in the context of the JHTDB and more broadly in scientific analysis cluster environments, which store large numerical simulation datasets. The evaluation is carried out from the point of view of the end user and includes all aspects of the process, from the initial request, to particle distribution, the I/O performed at each integration step, the integration time and finally the time to send the results back to the user. For the performance evaluation we use the datasets available in the JHTDB. Most of the experiments are run against the magnetohydrodynamics (MHD) dataset, which has 1024 time-steps and stores data on a 1024^3 spatial grid [98]. Additionally, we evaluate the different approaches on a forced isotropic turbulence dataset [9] and on

a homogeneous buoyancy-driven turbulence (HBDT) dataset [99]. We perform micro-benchmarks designed to examine the different characteristics of the particle tracking approaches and to offer a study at large scale, including large number of particles to be traced and long time spans. We also evaluate the approaches on workload extracted from the usage log of the JHTDB.

6.5.1 Experimental Setup

The experiments were carried out on the production database nodes of the JHTDB through a development Web-server/mediator hosting the Web-services. The nodes of the database cluster are part of the GrayWulf [16] and DataScope [17] clusters at JHU. The GrayWulf database nodes are 2.66 GHz dual quad-core Windows 2008 servers with SQL Server 2008 R2 and 24 GB of memory. Each node has 24 2TB SATA disks arranged as a set of four RAID-5 arrays. The DataScope database nodes are 3.47 GHz dual quad-core Windows 2008 servers with SQL Server 2008 R2 and 48 GB of memory. Each node has 24 2TB SATA disks arranged as 12 mirrored disk arrays. The MHD dataset is partitioned spatially across 4 GrayWulf nodes with one database per node and database files striped across the nodes and their associated disk arrays. The forced isotropic turbulence dataset is also partitioned across 4 GrayWulf nodes, but in this case there are 2 databases per node. The HBDT dataset is partitioned across 2 DataScope nodes with 4 databases per node.

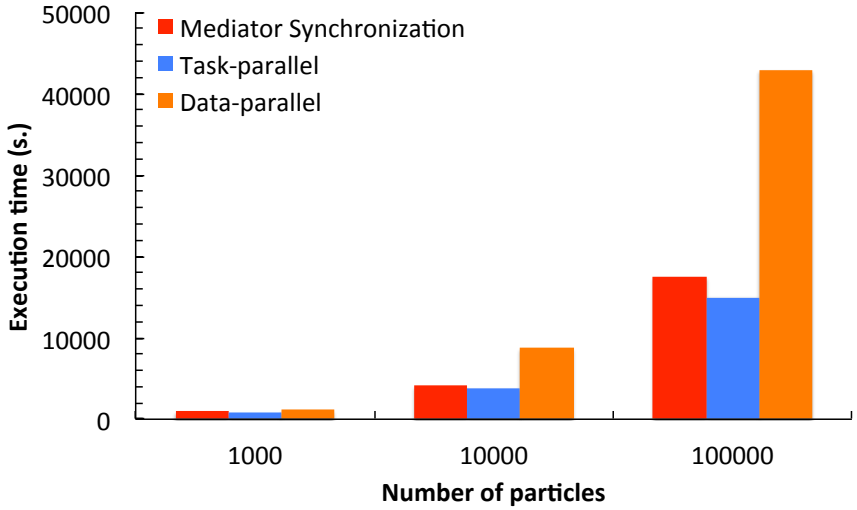


Figure 6.3: Execution times of the different particle tracking approaches for particles randomly distributed in the entire data volume with an integration step less than half of the temporal resolution of the data.

6.5.2 Micro-benchmarks

The first set of experiments that we perform evaluate the different approaches on workloads of varying number of particles, tracking them over a relatively large time span. Figure 6.3 shows the execution time for particle batches of 1000, 10,000 and 100,000 in the MHD dataset. For this set of experiments we run a single process per database node. The time span of each request is a fifth to a quarter of the entire time span of the simulation. This is a much larger time span than what is observed in typical user workload, but it allows particles to travel a large distance and can be used to examine the overall structure of the flow. Particles are distributed randomly in the entire data volume and an integral time-step of about one half to one third of the simulation time-step is used, which is the resolution at which user workloads typically operate. This results in a total of ∼1000 steps in the Runge-

Kutta integration scheme, which are carried out over 200 simulation time-steps. Each particle position is computed using 4th order Lagrange polynomial interpolation, which is again typical of the observed workload. Temporal interpolation is used by default, using data from 4 time-steps to evaluate every particle position.

As we can see from the results in the figure, the task-parallel approach outperforms the other two approaches by at least ∼15%. This is because it achieves a good balance between the retrieval of data for the entire batch of particles assigned to a process and allowing each process to advect particles independently. The data-parallel approach performs significantly worse. This is mainly due to the asynchronous nature of the particle tracking process in this case, which forces each process to retrieve data from multiple time-steps in order to advect the particles assigned to it during each integration step. Additionally, because of the lack of ghost regions around data boundaries, particles that are near the boundary have to be assigned to multiple processes for evaluation. Replicating data around the boundary is not an option in the JHTDB because of the large overhead that this introduces and the fact that many analysis queries do not need it.

The next set of experiments evaluates the three particle tracking approaches in different datasets, on different workloads and with different number of processes per database node (Figure 6.4). The number of processes varies mainly because of the different spatial distribution of the data for the different datasets. The workload for the forced isotropic turbulence dataset consists of 100,000 particles with an integration

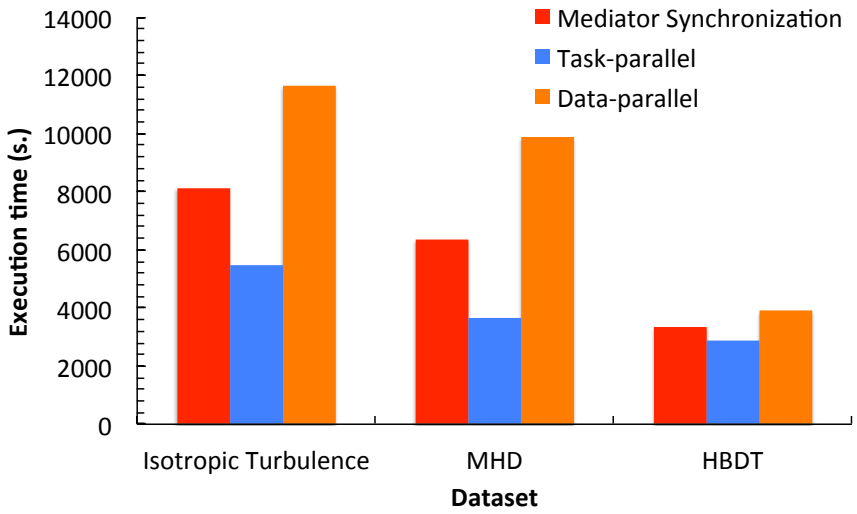


Figure 6.4: Execution times of the different particle tracking approaches for different workloads in three different datasets (forced isotropic turbulence, MHD and HBDT).

step of around one third the simulation time-step and traces particles over a time span of about a fifth of the entire time span stored in the database. Similarly as for the MHD dataset this results in a total of ∼1000 steps in the Runge-Kutta integration scheme. This dataset is distributed across 4 database nodes with 2 databases per node. We use 4 processes per database, for a total of 32 processes in these experiments. The workload for the MHD dataset is the 100,000 particle workload shown in Figure 6.3, but in this case we use 8 processes per database node, for a total of 32 processes. Finally, the workload for the HBDT dataset is 10,000 particles with similar integration step and time span as for the other two datasets. This dataset is distributed across 2 database nodes with 4 databases per node and we use 2 processes per database for the experiments.

The task-parallel approach performs best for all of the different datasets and parameterizations. In fact it is 75% faster than the mediator-synchronization approach

for the MHD dataset and 50% faster than the mediator-synchronization approach for the forced isotropic turbulence dataset. This is due to the fact that the task-parallel approach is able to scale better with increased process count as particle advection is carried out by each database process until completion of the entire batch and there is no movement of particles between the mediator and the database processes. The communication between the mediator and the database processes after each integration step does not scale as well with increased process count.

The data-parallel approach's relative performance is significantly better for the HBDT dataset as compared to the other two datasets. It is only 36% slower than the task-parallel approach, where as for the MHD dataset it is 2.7 times slower. This is due to the smaller size of the workload, which leads to particles making substantially fewer data-boundary crossings. In this case, there were around 220,000 such crossings, while the total number of crossings for the MHD dataset was 2.9 million. Moreover, the higher-end database nodes that host the HBDT dataset have faster CPUs, more memory and newer network cards.

6.5.3 User Workload

We evaluate the different particle tracking approaches on workload extracted from the usage logs of the JHTDB. Our analysis of the usage logs shows that most of the users' workload requests particle positions integrated over a relatively short time span compared to the time span of the entire simulation. Users also request integration

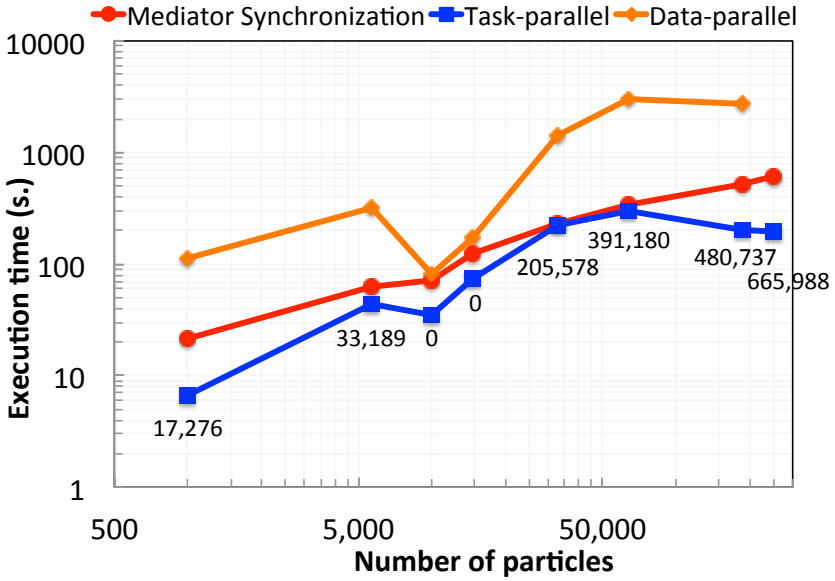


Figure 6.5: Performance of the different particle tracking approaches on workload extracted from the usage log of the JHTDB. Labels show the number of data boundary crossings.

time-steps that are a fraction of the separation between simulation time-steps stored in the database. For the experiments we have selected from the most commonly executed particle tracking queries with varying number of particles, from 1,000 to 250,000. Particles were integrated for a total of 100 Runge-Kutta integration steps over 20 simulation time-steps. The results are shown in Figure 6.5.

Overall the task-parallel approach performed the best and was up to 3 times better than synchronization at the mediator. This performance is even better than what we measured for the micro-benchmarks. This is because most user workload tends to be more localized, for example examining a region of interest, and in such cases particles do not cross data boundaries very often and each process performs more localized I/O. For each run we measured the number of times a particle crossed the

data boundaries of the process that it was assigned to (data points are labeled with these numbers in Figure 6.5).

User workloads distribute particles in the simulation domain with varying sparsity and localization characteristics. Some workloads distribute particles randomly or nearly randomly in the entire data volume as is the case for the 64,000 particles workload. Others confine particles to a much smaller region in space as is the case for the 185,000 and 250,000 particles workloads. The 10,000 and 14,641 particles workloads, which exhibit 0 particle crossings also distribute particles densely in space, bur moreover they happen to seed the initial particle positions near the center of the data region stored on a database node. Given the relatively short integration time, these "*particle clouds*" do not move far enough from the initial particle positions and do not cross into the data regions stored on adjacent nodes, resulting in the 0 crossings observed in these workloads.

While particle crossings have an impact on the overall execution time in some cases higher number of crossings does not necessarily lead to higher execution time. Another factor that has an impact on the execution time is the number of data blocks requested from the database. For example, the workload with 64,000 particles, which had 391,180 particle crossings had a higher execution time than the workloads with 185,000 and 250,000 particles. The 64,000 particles workload was distributing particles randomly in the entire data volume and subsequently almost all of the data blocks for each time-step had to be retrieved during particle advection. On the other

hand, the other two workloads were much more localized, resulting in requests for only about 1 to 2% of all data blocks of a time-step. The task-parallel approach is effectively able to take advantage of these workload characteristics, which reduces the overall running time. Synchronization at the mediator is much less sensitive to data locality as particles are continuously sent back and forth for reassignment and the size of the request is the dominating factor.

6.5.4 Scalability

Particle tracking for scientific analysis is a data-intensive task. Each integration steps requires the retrieval of data from 4 time-steps for the purposes of temporal interpolation and each particle requires between 64 and 512 data points from each time-step for the purposes of spatial interpolation. Our implementations show good scalability with increased process count per database as shown in Figure 6.6. We performed two sets of experiments against the MHD dataset. For the first set we issued particle tracking requests for 100,000 particles over a time span of 0.5 (out of a total simulation time span of 2.56) with an integral time-step of 0.001 (compared to a separation between simulation time-steps of 0.0025). This corresponded to 1,000 integration rounds (500 pairs of predictor and corrector steps in the Runge-Kutta scheme) over 200 simulation time-steps. The second set tracked 10,000 particles over a time span of 1.25 with an integral time-step 0.0025 equal to the separation between

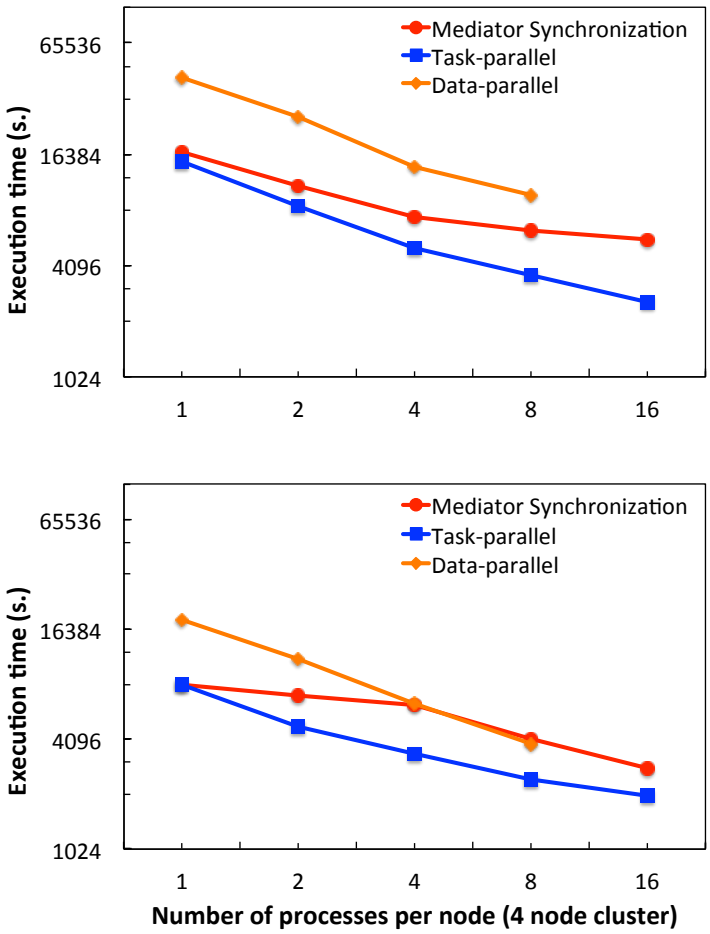


Figure 6.6: Performance of the different particle tracking approaches for 100,000 randomly distributed particles over 200 time-steps (left panel) and for 10,000 randomly distributed particles over 500 time-steps (right pannel) with different number of processes per database node in a 4 node cluster.

the stored simulation time-steps. This also corresponded to 1,000 integration rounds, but over 500 simulation time-steps.

The task-parallel and data-parallel approaches show better parallel efficiency than synchronization at the mediator, which is to be expected as they allow each process to perform particle integration independently. For the first set of experiments, the small integration time-step leads to slightly lower I/O time as database queries for

sequences of a few integration steps request data from the same time-steps, which makes the database cache very effective. For the second set of experiments even though the number of I/Os is the same every I/O has to bring data for at least one time-step from disk as the integral time-step is equal to the separation between stored simulation time-steps. This increased I/O time reduces the parallel efficiency.

The task-parallel approach to particle tracking has parallel efficiency over 50% up to 8 processes per node (which is the number of cores available on each node). This shows that this approach is not sensitive to finer data-partitioning and can be effectively utilized if the data are distributed across larger database clusters. The reason why the speedup diminishes with increased process count is that the data reside in the same database tables and on the same set of disks and while computation scales with increased process count, I/O contention increases and we observe an overall sub-linear speedup. If the data are distributed across more database nodes we would expect to see even better speedup as the I/O would be performed to different databases.

6.6 Related Work

Previous work on particle tracking and domain traversal approaches has focused on particle tracking for visualization of large datasets and in supercomputing environments. Ueng et al. [100] present a technique for out-of-core visualization of streamlines in large unstructured grids, which uses an octree to partition and restruc-

ture the raw data for fast retrieval. They focus on achieving optimal I/O performance with a small memory footprint. Yu et al. [95] describe a visualization technique that computes short integral segments called pathlets in time-varying vector fields. Their approach relies on a preprocessing step, which selects seed points and decomposes the data into blocks. This approach does not allow for user-specified seed points, which is important for the purposes of scientific analysis. Additionally, in our setting the data reside in relational databases and are used for a variety of analysis tasks in addition to particle tracking. Our goal is therefore to avoid preprocessing or restructuring the data.

Pugmire et al. [97] describe a hybrid approach that makes use of both static decomposition and loading data on demand. The process is coordinated by a master process that monitors load balance. Peterka et al. [94] investigate static and dynamic partitioning strategies and show that simple round-robin partitioning often outperforms dynamic partitioning. These approaches were demonstrated on up to 512 Cray XT cores and 32 K Blue Gene/P cores, respectively.

Kendall et al. [93] introduce a system called *DStep* aimed at simplifying domain traversal techniques. DStep utilizes different workers to perform domain traversal tasks. Workers are partitioned into groups with a communicator worker assigned to each group responsible for inter-group communication. The shift to group-based communication for domain traversal techniques is driven by the need to alleviate the network congestions induced by asynchronous communication. Our study also

indicates that a purely data-parallel asynchronous approach may suffer performance penalties, however mostly due to its I/O pattern. In the DStep system the large distributed memory of a supercomputer is leveraged to perform all I/O upfront, which is not feasible for smaller analysis clusters.

6.7 Discussion

We have presented a study of particle tracking approaches, which advect particles for scientific analysis purposes in an open simulation laboratory, the JHTDB. Particle tracking is both data and computationally intensive task and the approaches that we compare are implemented with the goal of minimizing overall I/O. We examine data and task parallelization techniques as well as an approach that synchronizes execution at the mediator level. We evaluate these techniques on datasets 100s of TB in size in a live production environment. We observe that the task-parallel approach to particle tracking outperforms the other approaches by up to a factor of 3 for a variety of workloads and parameterizations. It achieves over 50% parallel efficiency and is able to scale the execution by performing the entire advection process on the database nodes of the analysis cluster environment.

Our study highlights the trade-offs between the batch execution of database queries for the retrieval of data necessary during each integration step and asynchronous processing and assignment of particles as they traverse the data volume and

cross data boundaries. We find that the batch execution of particles, retrieving data from the same set of time-steps as needed for temporal interpolation during each integration step outweighs the benefits of asynchronous processing. In order to support asynchronous processing the I/O pattern changes fundamentally and has to allow for the retrial of data blocks from multiple different time-steps, which results in slower I/O.

In the future, we plan to extend the analysis capabilities of the JHTDB and provide more built-in tools for the study of fluid dynamics in the simulation laboratory. One such capability in the context of particle tracking is the computation of the Finite-Time Lyapunov Exponent (FTLE) field. In recent years, significant efforts have been devoted to studying flow using visualizations of FTLEs [101, 102]. Nouanesengsy et al. [103] describe a framework and a parallelization strategy to trace the massive number of particles necessary for FTLE computation using a supercomputer. We plan to investigate similar techniques that can be adapted to the scientific analysis cluster environment.

Particle tracking is not yet available for one of the datasets recently added to the JHTDB, the channel flow dataset. This is because this dataset simulates turbulent flow between walls in a channel and the velocity goes to 0 near the walls. Our turbulence-research collaborators are still investigating the proper way to handle particles that approach the walls of the channel. We plan to extend the particle tracking capabilities of the JHTDB to the channel flow dataset in the near future.

Chapter 7

Impact of the JHTDB

As of May 2015 the JHTDB has served queries to nearly 13 trillion data points. This serves as a testament to the demand for such high-quality data and their potential for reuse. Each of the four datasets stored in the JHTDB currently has on the order of a trillion space-time data points. Storing these data in an open simulation laboratory preserves the computational effort used to generate them, allows for repeatability of experiments and verification of results and enables users to come back to the same place in space and time. The continued exploration of the same place in space and time produces the same exact data, but users can look at it from different angles, requesting derivatives, spatially filter the data or compute more complicated quantities. This builds physical intuition about the nonlinear dynamics of the complex flow phenomena and can lead to new discoveries.

The availability of an open simulation laboratory for the study of turbulent flows

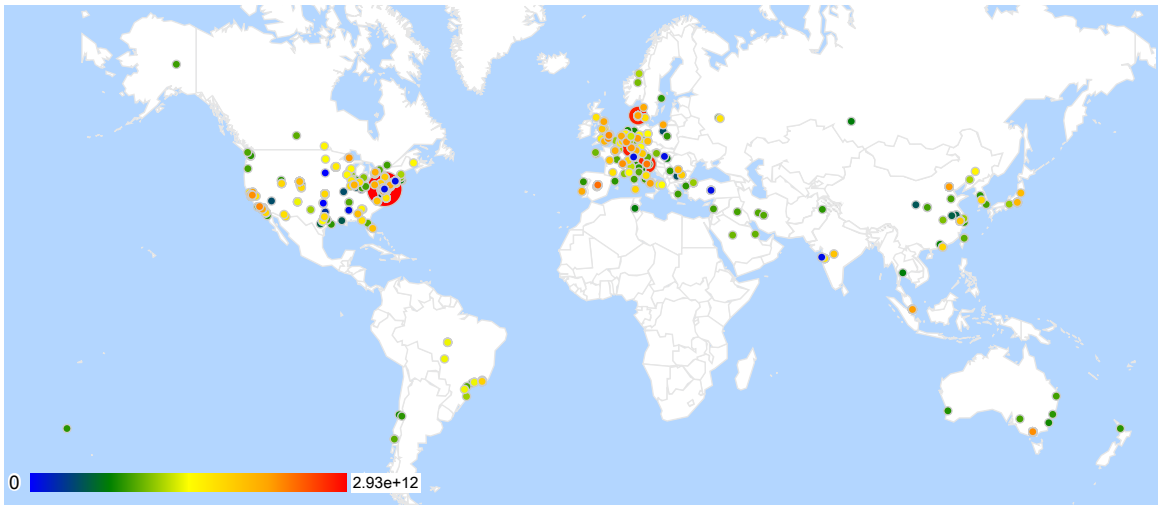
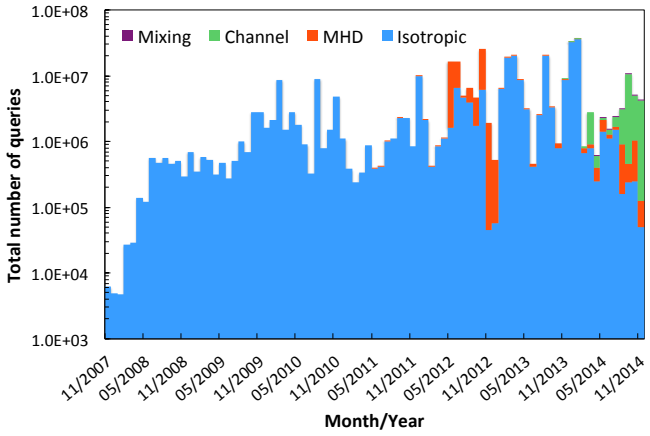
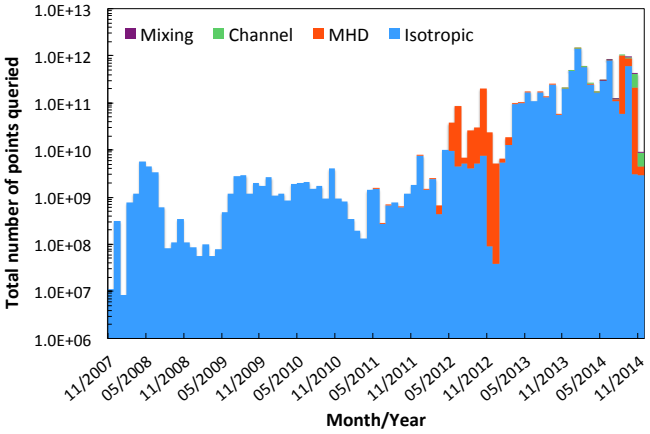


Figure 7.1: Origin location of queries to the JHTDB. Each IP address location is colored based on the total number of points queried in log scale.

has already had a major impact on turbulence research and education. The JHTDB has over 150 registered users and queries have been made from over 1500 distinct ip addresses across the globe (Figure 7.1). Data from the service have been used fully or partially in multiple research projects resulting in at least 40 peer-reviewed publications, including in high-profile venues such as Nature [4], Proceedings of the National Academy of Sciences [104], and Physical Review Letters [89]. The impact of the JHDTB has extended to the classroom as well. The laboratory has been used in various classes and student workshops around the world (e.g. the "Tutorial School on Fluid Dynamics: Topics in Turbulence" held at the University of Maryland College Park - see http://www2.cscamm.umd.edu/programs/trb10/, to be repeated in the summer of 2015).



(a) Number of queries

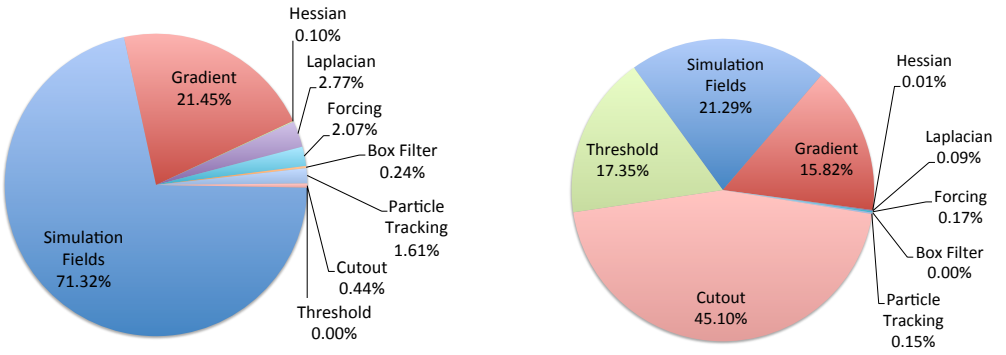


(b) Number of data points queried

Figure 7.2: Semi-log plot of the number of queries and number of data points queried from the JHTDB per year.

7.1 Analysis of JHTDB Usage

JHTDB's usage has grown steadily over the years with the addition of new datasets and expanded capabilities and performance. Figure 7.2(a) shows the total number of queries submitted per year since the JHTDB's inception in 2007 and the total number of points queried is shown in Figure 7.2(b). As we can see there has been an exponential growth in the total number of data points queried over the past several



(a) Percentage of queries per query type

(b) Percentage of data points queried per query type

Figure 7.3: Distribution of JHTDB queries and number of data points queried per query type.

years, mainly due to new access methods, increased number of analysis functionalities and the addition of new datasets. The MHD dataset was released in 2011, the channel flow dataset became public in 2013 and the mixing dataset was released in 2014. In 2011 the particle tracking functionality was implemented and made available, while in 2012 the data cutout service went online. These new functionalities have greatly increased the utility of the service. The data-intensive iterative process used to track particles is much more efficiently executed near the data. On the other hand, the cutout service has provided an easy and efficient way for researchers to obtain the raw simulation data if they so choose.

As Figure 7.3(a) shows, most queries submitted and evaluated by the JHTDB request the simulation field parameters and their spatial derivatives. The number of requests made through the cutout service is significantly smaller. However, in terms of the actual size of the requests and the total amount of data retrieved, the

cutout requests are a much larger slice of the pie as seen in Figure 7.3(b). The recently introduced threshold queries are another example of an extremely data-intensive task, which is efficiently performed by the database cluster. A single request for the number of points above a particular threshold usually examines up to the entire 1024^3 or larger data volume of a simulation time-step. There have been only $\sim$2000 such queries executed so far, and yet over 1 trillion points have been examined by this query type alone.

Chapter 8

Conclusions & Future Work

The development of architectures, systems and algorithms that can handle and utilize large amounts of complex data is of paramount importance for scientific discovery. Today's supercomputers generate as much data as traditional experimentation and observation approaches. Some of the largest fluid dynamics simulations produce petabytes of data [8]. Algorithmic readiness for the management of such large datasets has been identified as a major challenge at two NSF workshops on cyber-fluid dynamics [105, 106]. The Johns Hopkins Turbulence Databases have been built with the goal of addressing some of the challenges with the curation of and providing public access to landmark numerical simulation datasets of fluid dynamics. In this dissertation, I have described methods and software architectures that my collaborators and I have developed to enable efficient exploration and analysis of landmark datasets in open simulation laboratories such as the JHTDB.

We have developed an I/O streaming partial-sums method for the evaluation of batch-queries performing decomposable kernel computations at a large number of target locations (Section 3). The method evaluates a batch-query by means of a single sequential pass over the data and effectively exploits data-sharing between individual requests. It outperforms direct approaches or approaches derived from the simulation code by over an order of magnitude. For the largest of spatial filtering queries we have introduced a complementary method of execution based on an intermediate summed-volumes dataset (Section 4). Box filtering can be performed by looking up only 8 data points in this summed-volumes dataset. The two methods are incorporated into a query-processing framework, which dynamically chooses the best method based on the workload characteristics.

Executing threshold queries on entire time-steps of the simulation allows researches to examine large amounts of data and quickly narrow down on some of the most interesting regions in the dataset. We have developed a system for the evaluation of threshold queries of derived fields in the JHTDB (Section 5). We have extended the database cluster with an application-aware semantic cache for the results of threshold queries. The query results are stored persistently on SSDs attached to each database node and can be used to answer subsequent queries. Cache hits reduce the running time of threshold queries from minutes to several seconds.

The study of the Lagrangian description of turbulence makes extensive use of particle tracking techniques. We have developed a task-parallel method for the advection

of particles along their trajectories in the simulated flow (Section 6). The method outperforms alternative approaches by advecting a batch of particles together and retrieving the velocity data necessary for the advection using the I/O streaming method for batch-queries.

8.1 Future Work

Our vision for the way forward is a tighter integration and coupling between numerical simulations and simulation laboratories. The in situ analysis performed during the simulation will automatically generate a database to verify and repeat the analysis or to further examine the data. Conversely, analysis queries to an existing database will launch simulations that refine the data or explore new science questions. This will require the creation of two-way connections between in situ simulation analyses and archival databases with the goal of unifying simulation and analysis. In this way, a tradeoff between computation and storage can be leveraged.

Not all simulations need to be archived, but landmark simulations can be preserved in large parts until they become obsolete and the storage needs to be reclaimed for the next landmark simulation. If a simulations output is larger than the available storage, it can be stored at a lower resolution, with regions of interest refined or resimulated on demand. Analysis queries can execute against the stored data, launch a new computation, or perform a combination of the two.

The Web-services approach to archived numerical simulation datasets provides public access to high quality simulation data to anyone with an Internet connection. The Web-services methods can be called from any modern programming language and we provide C, Fortran and Matlab client libraries for the JHTDB. The evaluation of each query submitted through a Web-service call is carried out on the nodes of the database cluster by means of a stored procedure or a user-defined function that has been implemented and deployed to handle these requests. This allows us to fine-tune the execution of these procedures and handle all requests transparently to the user. However, this approach also has drawbacks. Adding new functionality means adding to a long list of Web-service calls and requires substantial implementation effort. For example, in the case of threshold queries the stored procedure performing the evaluation must have an implementation for each derived field of interest even though the execution is handled by the same Web-service call.

In the future, we plan to develop declarative and graphical user interfaces that will allow users to combine existing building blocks and perform computations that have not been explicitly implemented. Additionally, we plan on deploying a server-side computing environment for users similar to the CasJobs service for the Sloan Digital Sky Survery [107]. In such an environment users can direct output of their batch-queries to personal databases, called MyDB. This will allow for much greater flexibility in the type of computations that can be performed in addition to substantially decreasing the network overhead. In the case of particle tracking, this will allow

users to store trajectories along with additional quantities at each particle position without having to transfer large amounts of data over the network. We also plan to provide server-side visualization and rendering capabilities that will allow users to visualize particle trajectories and other data in an interactive environment.

The introduction of an application-aware cache for the results of threshold queries lays the groundwork for the creation of a landmark database. Such a database can store the locations of the highest vorticity regions in the dataset or more broadly regions of interest and their associated statistics. Similarly, the spatial-filtering framework can be utilized to generate a dataset hierarchy, where some datasets or parts thereof can be stored at different spatial resolutions. The multi-resolution, space-time datasets can be used to investigate the cascade of kinetic energy from large to small scales.

The JHTDB has already had a transformative impact on the study of fluid dynamics and our understanding of turbulence flows. It is the manifestation of the "remote immersive analysis" approach to computational science. It provides public access to high-quality turbulence simulation data along with built-in analysis functionality. Scientists and the public as a whole can immerse sensors in the simulated flows, study their evolution or the flow parameters. This can and has been done by researchers from around the world. Derived and landmark database functionalities coupled with on-demand visualization capabilities and better integration with hardware will allow for interacting with fluid dynamics data in new and different ways.

Bibliography

[1] S. Pope, *Turbulent Flows*. Cambridge University Press, 2000.

[2] M. Lee, R. Ulerich, N. Malaya, and R. D. Moser, "Experiences from Leadership Computing in Simulations of Turbulent Fluid Flows," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 24–31, Sep. 2014.

[3] K. Kanov, R. Burns, C. Lalescu, and G. Eyink, "The Johns Hopkins Turbulence Databases: An Open Simulation Laboratory for Turbulence Research," *Computing in Science & Engineering*, Sept. 2015, to be published.

[4] G. Eyink, E. Vishniac, C. Lalescu, H. Aluie, K. Kanov, K. Bürger, R. Burns, C. Meneveau, and A. Szalay, "Flux-freezing breakdown in high-conductivity magnetohydrodynamic turbulence." *Nature*, vol. 497, no. 7450, pp. 466–9, 2013.

[5] B. Luethi, M. Holzner, and A. Tsinober, "Expanding the q–r space to three dimensions," *Journal of Fluid Mechanics*, vol. 641, pp. 497–507, 2009.

[6] A. G. Gungor and S. Menon, "A new two-scale model for large eddy simulation

of wall-bounded flows," *Progress in Aerospace Sciences*, vol. 46, no. 1, pp. 28–45, 2010.

[7] J. M. Lawson and J. R. Dawson, "A scanning piv method for fine-scale turbulence measurements," *Experiments in Fluids*, vol. 55, no. 12, pp. 1–19, 2014.

[8] M. Lee, N. Malaya, and R. D. Moser, "Petascale direct numerical simulation of turbulent channel flow on up to 786k cores," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013, pp. 61:1–61:11.

[9] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns, S. Chen, A. Szalay, and G. Eyink, "A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence," *Journal of Turbulence*, vol. 9, p. N31, 2008.

[10] E. Perlman, R. Burns, Y. Li, and C. Meneveau, "Data Exploration of Turbulence Simulations Using a Database Cluster," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07, 2007, pp. 23:1–23:11.

[11] F. N. Fritsch and R. E. Carlson, "Monotone piecewise cubic interpolation," *SIAM Journal on Numerical Analysis*, vol. 17, no. 2, pp. 238–246, 1980.

[12] Johns Hopkins Turbulence Databases. [Online]. Available: http://turbulence.pha.jhu.edu/

BIBLIOGRAPHY

[13] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," in *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '85, 1985, pp. 268–279.

[14] P. A. Boncz, W. Quak, and M. L. Kersten, "Monet and its geographic extensions: A novel approach to high performance GIS processing," in *In Proceedings of the International Conference on Extending Database Technology*, ser. EDBT '96, 1996, pp. 147–166.

[15] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: a column-oriented DBMS," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, 2005, pp. 553–564.

[16] A. S. Szalay, G. Bell, J. Vandenberg, A. Wonders, R. Burns, D. Fay, J. Heasley, T. Hey, M. Nieto-Santisteban, A. Thakar, C. van Ingen, and R. Wilton, "GrayWulf: Scalable Clustered Architecture for Data Intensive Computing," in *Proceedings of the 42nd Hawaii International Conference on System Sciences*, ser. HICSS '09, 2009, pp. 1–10.

[17] DataScope. [Online]. Available: http://idies.jhu.edu/datascope

[18] The HDF Group. (1997-2015) Hierarchical Data Format, version 5. [Online]. Available: http://www.hdfgroup.org/HDF5/

[19] K. Kanov, E. Perlman, R. Burns, Y. Ahmad, and A. Szalay, "I/O Streaming Evaluation of Batch Queries for Data-intensive Computational Turbulence," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 29:1–29:10.

[20] R. J. Purser and L. M. Leslie, "An Efficient Interpolation Procedure for High-Order Three-Dimensional Semi-Lagrangian Models," *Monthly Weather Review*, vol. 119, pp. 2492–+, 1991.

[21] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07, 2007, pp. 1029–1040.

[22] P. Agrawal, D. Kifer, and C. Olston, "Scheduling shared scans of large data files," *Proceedings of VLDB*, vol. 1, no. 1, pp. 958–969, August 2008.

[23] X. Wang, R. C. Burns, and T. Malik, "Liferaft: Data-driven, batch processing for the exploration of scientific databases," in *Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research*, ser. CIDR '09, 2009.

[24] X. Wang, E. Perlman, R. Burns, T. Malik, T. Budavári, C. Meneveau, and A. Szalay, "Jaws: Job-aware workload scheduling for the exploration of turbulence simulations," in *Proceedings of the 2010 ACM/IEEE International Con-*

ference for High Performance Computing, Networking, Storage and Analysis, ser. SC '10, 2010, pp. 1–11.

[25] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain, "All-pairs: An abstraction for data-intensive computing on campus grids," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, pp. 33–46, January 2010.

[26] L. Yi, C. Moretti, S. Emrich, K. Judd, and D. Thain, "Harnessing parallelism in multicore clusters with the all-pairs and wavefront abstractions," in *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '09, 2009, pp. 1–10.

[27] J.-B. Yu and D. J. DeWitt, "Query pre-execution and batching in Paradise: A two-pronged approach to the efficient processing of queries on tape-resident raster images," in *Proceedings of the Ninth International Conference on Scientific and Statistical Database Management*, ser. SSDBM '97, 1997, pp. 64–78.

[28] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable performance for unpredictable workloads," *Proceedings of Very Large Data Bases*, vol. 2, no. 1, August 2009.

[29] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally, "Executing irregular scientific applications on stream architectures," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ser. ICS '07, 2007, pp. 93–104.

BIBLIOGRAPHY

[30] X. Yang, J. Du, X. Yan, and Y. Deng, "Matrix-based streamization approach for improving locality and parallelism on ft64 stream processor," *J. Supercomput.*, vol. 47, no. 2, pp. 171–197, February 2009.

[31] J. W. Romein, P. C. Broekema, E. van Meijeren, K. van der Schaaf, and W. H. Zwart, "Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer," in *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '06, 2006, pp. 59–66.

[32] L. Neugebauer, "Optimization and evaluation of database queries including embedded interpolation procedures," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '91, 1991, pp. 118–127.

[33] R. H. Wolniewicz and G. Graefe, "Algebraic optimization of computations over scientific databases," in *Proceedings of the 19th International Conference on Very Large Data Bases*, ser. VLDB '93, 1993, pp. 13–24.

[34] A. Deshpande and S. Madden, "MauveDB: supporting model-based user views in database systems," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06, 2006, pp. 73–84.

[35] S. Grumbach, P. Rigaux, and L. Segoufin, "Manipulating interpolated data is easier than you thought," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00, 2000, pp. 156–165.

[36] A. Thiagarajan and S. Madden, "Querying continuous functions in a database system," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, 2008, pp. 791–804.

[37] K. Kanov, R. Burns, G. Eyink, C. Meneveau, and A. Szalay, "Data-intensive Spatial Filtering in Large Numerical Simulation Datasets," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 1–9.

[38] F. C. Crow, "Summed-area Tables for Texture Mapping," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '84, 1984, pp. 207–212.

[39] A. Leonard, "Energy Cascade in Large-Eddy Simulations of Turbulent Fluid Flows," *Advances in Geophysics*, vol. 18, p. A237, 1974.

[40] C. Meneveau and J. Katz, "Scale-Invariance and Turbulence Models for Large-Eddy Simulation," *Annual Review of Fluid Mechanics*, vol. 32, no. 1, pp. 1–32, 2000.

[41] S. B. Pope, "Lagrangian pdf methods for turbulent flows," *Annual Review of Fluid Mechanics*, vol. 26, no. 1, pp. 23–63, 1994.

[42] T. Lund, "The use of Explicit Filters in Large Eddy Simulation," *Computers & Mathematics with Applications*, vol. 46, no. 4, pp. 603–616, 2003.

BIBLIOGRAPHY

[43] J. C. Del Álamo, J. Jiménez, P. Zandonade, and R. D. Moser, "Self-similar Vortex Clusters in the Turbulent Logarithmic Region," *Journal of Fluid Mechanics*, vol. 561, pp. 329–358, Aug. 2006.

[44] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., 2006.

[45] T. Randen and J. Husoy, "Filtering for texture classification: a comparative study," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21, no. 4, pp. 291 –310, Apr. 1999.

[46] J. Ebling and G. Scheuermann, "Clifford Fourier Transform on Vector Fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 4, pp. 469–479, Jul. 2005.

[47] O. Fialka and M. Cadik, "FFT and Convolution Performance in Image Filtering on GPU," in *Proceedings of the Conference on Information Visualization*, ser. IV '06, 2006, pp. 609–614.

[48] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz, "Distributed Processing of Very Large Datasets with DataCutter," *Parallel Comput.*, vol. 27, no. 11, pp. 1457–1478, Oct. 2001.

[49] B. Allcock, I. Foster, V. Nefedova, A. Chervenak, E. Deelman, C. Kesselman, J. Lee, A. Sim, A. Shoshani, B. Drach, and D. Williams, "High-performance

Remote Access to Climate Simulation Data: a Challenge Problem for Data Grid Technologies," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01, 2001, pp. 46–46.

[50] Climate Data Analysis Tools. [Online]. Available: http://www2-pcmdi.llnl.gov/cdat

[51] L. A. Freitag and R. M. Loy, "Adaptive, Multiresolution Visualization of Large Data Sets using a Distributed MemoryOctree," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99, 1999, p. 60.

[52] K. liu Ma, "Parallel rendering of 3D AMR data on the SGI/Cray T3E," in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '99, 1999, pp. 138–.

[53] T. W. Crockett and T. Orloff, "A MIMD Rendering Algorithm for Distributed Memory Architectures," in *Proceedings of the 1993 Symposium on Parallel Rendering*, ser. PRS '93, 1993, pp. 35–42.

[54] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi, "Passion: Optimized I/O for Parallel Applications," *Computer*, vol. 29, no. 6, pp. 70–78, Jun. 1996.

[55] A. P. Marathe and K. Salem, "Query Processing Techniques for Arrays," in *Pro-*

ceedings of the 1999 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '99, 1999, pp. 323–334.

[56] P. G. Brown, "Overview of scidb: large scale array storage, processing and analysis," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, 2010, pp. 963–968.

[57] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez, "The datapath system: a data-centric analytic processing engine for large data warehouses," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, 2010, pp. 519–530.

[58] K. Kanov, R. C. Burns, and C. C. Lalescu, "Efficient evaluation of threshold queries of derived fields in a numerical simulation database," in *Proceedings of the 18th International Conference on Extending Database Technology*, ser. EDBT '15, 2015, pp. 301–312.

[59] A. J. G. Hey, S. Tansley, and K. M. Tolle, *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[60] The Sloan Digital Sky Survey. [Online]. Available: http://www.sdss.org/

[61] The Millennium Simulation. [Online]. Available: http://www.mpa-garching.mpg.de/millennium/

[62] R. Burns, K. Lillaney, D. R. Berger, L. Grosenick, K. Deisseroth, R. C. Reid,

W. G. Roncal, P. Manavalan, D. D. Bock, N. Kasthuri, M. Kazhdan, S. J. Smith, D. Kleissas, E. Perlman, K. Chung, N. C. Weiler, J. Lichtman, A. S. Szalay, J. T. Vogelstein, and R. J. Vogelstein, "The open connectome project data cluster: Scalable analysis and vision for high-throughput neuroscience," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM, 2013, pp. 27:1–27:11.

[63] J. C. Lopez, D. R. O'Hallaron, and T. Tu, "Big Wins With Small Application-aware Caches," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, ser. SC '04, 2004, pp. 20–.

[64] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement," in *Proceedings of the 22th International Conference on Very Large Data Bases*, ser. VLDB '96, 1996, pp. 330–341.

[65] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "The Multidimensional Database System RasDaMan," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '98, 1998, pp. 575–577.

[66] M. Stonebraker, J. Becla, D. J. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik, "Requirements for Science Data Bases and SciDB," in *Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research*, ser. CIDR '09, 2009.

[67] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes, "SciQL: Bridging the Gap Between Science and Relational DBMS," in *Proceedings of the 15th Symposium on International Database Engineering & Applications*, ser. IDEAS '11, 2011, pp. 124–133.

[68] P. Baumann, "A Database Array Algebra for Spatio-Temporal Data and Beyond," in *Proceedings of the 4th International Workshop on Next Generation Information Technologies and Systems*, ser. NGIT '99, 1999.

[69] MonetDB. [Online]. Available: http://monetdb.cwi.nl/

[70] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A Survey of Top-k Query Processing Techniques in Relational Database Systems," *ACM Comput. Surv.*, vol. 40, no. 4, pp. 11:1–11:58, Oct. 2008.

[71] P. M. Deshpande, D. P, and K. Kummamuru, "Efficient Online top-K Retrieval with Arbitrary Similarity Measures," in *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '08, 2008, pp. 356–367.

[72] D. Xin, J. Han, and K. C. Chang, "Progressive and Selective Merge: Computing Top-k with Ad-hoc Ranking Functions," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07, 2007, pp. 103–114.

[73] W.-T. Balke, W. Nejdl, W. Siberski, and U. Thaden, "Progressive Distributed Top-k Retrieval in Peer-to-Peer Networks," in *Proceedings of the 21st International Conference on Data Engineering*, ser. ICDE '05, 2005, pp. 174–185.

[74] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis, "On Efficient Top-k Query Processing in Highly Distributed Environments," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, 2008, pp. 753–764.

[75] P. Cao and Z. Wang, "Efficient top-K Query Calculation in Distributed Networks," in *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '04, 2004, pp. 206–215.

[76] S. Chaudhuri, L. Gravano, and A. Marian, "Optimizing Top-k Selection Queries over Multimedia Repositories," *IEEE Trans. on Knowl. and Data Eng.*, vol. 16, no. 8, pp. 992–1009, Aug. 2004.

[77] U. Güntzer, W.-T. Balke, and W. Kießling, "Optimizing Multi-Feature Queries for Image Databases," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00, 2000, pp. 419–428.

[78] A. Marian, N. Bruno, and L. Gravano, "Evaluating Top-k Queries over Web-accessible Databases," *ACM Trans. Database Syst.*, vol. 29, no. 2, pp. 319–362, Jun. 2004.

[79] S. Michel, P. Triantafillou, and G. Weikum, "KLEE: A Framework for Distributed Top-k Query Algorithms," in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, 2005, pp. 637–648.

[80] K. Zhao, Y. Tao, and S. Zhou, "Efficient Top-k Processing in Large-scaled Distributed Environments," *Data Knowl. Eng.*, vol. 63, no. 2, pp. 315–335, Nov. 2007.

[81] Q. Ren, M. H. Dunham, and V. Kumar, "Semantic Caching and Query Processing," *IEEE Trans. on Knowl. and Data Eng.*, vol. 15, no. 1, pp. 192–210, Jan. 2003.

[82] J. Aßfalg, H.-P. Kriegel, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz, "Similarity Search on Time Series based on Threshold Queries," in *Proceedings of the 10th International Conference on Advances in Database Technology*, ser. EDBT'06, 2006, pp. 276–294.

[83] L. Sidirourgos, M. L. Kersten, and P. A. Boncz, "Sciborq: Scientific data management with bounds on runtime and quality," in *Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research*, ser. CIDR '11, 2011, pp. 296–301.

[84] S. Nirkhiwale, A. Dobra, and C. Jermaine, "A sampling algebra for aggregate estimation," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1798–1809, Sep. 2013.

BIBLIOGRAPHY

[85] H. Andrade, T. Kurc, A. Sussman, and J. Saltz, "Active semantic caching to optimize multidimensional data analysis in parallel and distributed environments," *Parallel Computing*, vol. 33, no. 7-8, pp. 497–520, Aug. 2007.

[86] K. Kanov and R. Burns, "Particle tracking in open simulation laboratories," submitted for publication, 2015.

[87] H. Yu, K. Kanov, E. Perlman, J. Graham, E. Frederix, R. Burns, A. Szalay, G. Eyink, and C. Meneveau, "Studying lagrangian dynamics of turbulence using on-demand fluid particle tracking in a public turbulence database," *Journal of Turbulence*, vol. 13, p. N12, 2012.

[88] D. Benveniste and T. Drivas, "Asymptotic results for backwards two-particle dispersion in a turbulent flow," *Physical Review E*, vol. 89, no. 4, p. 041003, 2014.

[89] J. Jucha, H. Xu, A. Pumir, and E. Bodenschatz, "Time-reversal-symmetry breaking in turbulence," *Physical Review Letters*, vol. 113, no. 5, p. 054501, 2014.

[90] G. Eyink and D. Benveniste, "Diffusion approximation in turbulent two-particle dispersion," *Physical Review E*, vol. 88, no. 4, p. 041001, 2013.

[91] K. Gustavsson, J. Einarsson, and B. Mehlig, "Tumbling of small axisymmetric

particles in random and turbulent flows," *Physical Review Letters*, vol. 112, no. 1, p. 014501, 2014.

[92] M. Raiola, S. Discetti, and A. Ianiro, "On piv random error minimization with optimal pod-based low-order reconstruction," *Experiments in Fluids*, vol. 56, no. 4, 2015.

[93] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson, "Simplified parallel domain traversal," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 10:1–10:11.

[94] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang, "A study of parallel particle tracing for steady-state and time-varying flow fields," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11, 2011, pp. 580–591.

[95] H. Yu, C. Wang, and K.-L. Ma, "Parallel hierarchical visualization of large time-varying 3d vector fields," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07, 2007, pp. 24:1–24:12.

[96] M. Holzner, A. Liberzon, N. Nikitin, B. Lüthi, W. Kinzelbach, and A. Tsinober, "A lagrangian investigation of the small-scale features of turbulent entrainment through particle tracking and direct numerical simulation," *Journal of Fluid Mechanics*, vol. 598, pp. 465–475, 3 2008.

[97] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber, "Scalable computation of streamlines on very large datasets," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 16:1–16:12.

[98] H. Aluie, G. Eyink, E. Vishniac, S. Chen, K. Kanov, R. Burns, C. Meneveau, and A. Szalay, "Forced MHD turbulence data set (2013)," Available at http://turbulence.pha.jhu.edu/docs/README-MHD.pdf.

[99] D. Livescu, C. Canada, K. Kanov, Burns, R. & IDIES staff, and J. Pulido, "Homogeneous Buoyancy driven turbulence data set (2014)," Available at http://turbulence.pha.jhu.edu/docs/README-HBDT.pdf.

[100] S.-K. Ueng, C. Sikorski, and K.-L. Ma, "Out-of-core streamline visualization on large unstructured meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, no. 4, pp. 370–380, 1997.

[101] G. Haller, "Finding finite-time invariant manifolds in two-dimensional velocity fields," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 10, no. 1, pp. 99–108, 2000.

[102] Haller, G., "Distinguished material surfaces and coherent structures in three-dimensional fluid flows," *Physica D*, vol. 149, no. 4, pp. 248–277, 2001.

[103] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka, "Parallel

particle advection and FTLE computation for time-varying flow fields," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 61:1–61:11.

[104] H. Xu, A. Pumir, G. Falkovich, E. Bodenschatz, M. Shats, H. Xia, N. Francois, and G. Boffetta, "Flight–crash events in turbulence," *Proceedings of the National Academy of Sciences*, vol. 111, no. 21, pp. 7558–7563, 2014.

[105] P. Yeung, R. Moser, M. Plesniak, C. Meneveau, S. Elghobashi, and C. Aidun, "Final Report to the National Science Foundation on NSF Workshop on Cyber-Fluid Dynamics: New Frontiers in Research and Education," 2007.

[106] "Fluid Dynamics Community Workshop: The Development of Fluid Mechanics Community Software and Data Resources," 2010. [Online]. Available: http://fluids.ices.utexas.edu/

[107] N. Li and A. R. Thakar, "CasJobs and MyDB: A Batch Query Workbench," *Computing in Science and Engineering*, vol. 10, no. 1, pp. 18–29, 2008.

Vita

Kalin Kanov was born in Dimitrovgrad, Bulgaria on October 11th, 1982. He graduated with distinction from the University of Virginia in 2006 with a B.A. degree in Astronomy and Physics and a minor in Computer Science. He was awarded the Limber award, given to the most outstanding Astronomy graduate of the class of 2006. After graduating from UVA he interned at NASA's Goddard Space Flight Center and worked at Perrin Quarles Associates on the development of the Emissions Collection and Monitoring Plan System for the U.S. EPA Clean Air Markets Division.

Kalin enrolled in the Computer Science Ph.D. program at Johns Hopkins University in 2008. As an entering Ph.D. student he was awarded the Pond Fellowship by the faculty of the Computer Science Department. During internships at Los Alamos National Laboratory and Google he worked on scientific database systems and evaluation techniques for complex arithmetic expressions over dataset features partitioned into columns. His research has focused on the development of methods for the efficient evaluation of batch-queries for large numerical simulation datasets.